

# Rappresentazione della scacchiera

## PARTE 2

*La rappresentazione della scacchiera è il primo passo da analizzare prima di cominciare a sviluppare il motore vero e proprio*

di **Andrea Lanza** > [alanza@infomedia.it](mailto:alanza@infomedia.it)

**I**l primo approccio che viene in mente ad un neofita che si appresta a rappresentare la scacchiera interna per il suo motore è una rappresentazione a matrice 8x8 dove, ogni casella viene rappresentata con un valore numerico.

Ad esempio: 13 rappresenta una casella vuota, 1 il Pedone bianco, 2 il Pedone nero ecc...

Questa rappresentazione è di per se la più semplice da implementare ma presenta diversi svantaggi perché il programma deve continuamente aggiornare i due indici "Riga" e "Colonna" rendendo il tutto molto lento e macchinoso. Chi vuole utilizzare questa rappresentazione sappia che prima o poi si troverà nella necessità di riscrivere completamente il programma sostituendo a questa rappresentazione una più veloce ed efficace, come potrebbe essere la rappresentazione 12x10.

In questo caso la scacchiera è rappresentata da un array di 120 elementi (come illustrato in **Figura 1**), assumendo ad esempio che le caselle non valide (il bordo) abbiano valore 0.

Questa rappresentazione semplifica e velocizza la generazione delle mosse in quanto, con delle veloci operazioni logiche, è più semplice verificare quando un pezzo va a cadere sul bordo della scacchiera.

Possibili valori per rappresentare le caselle ed i pezzi possono essere:

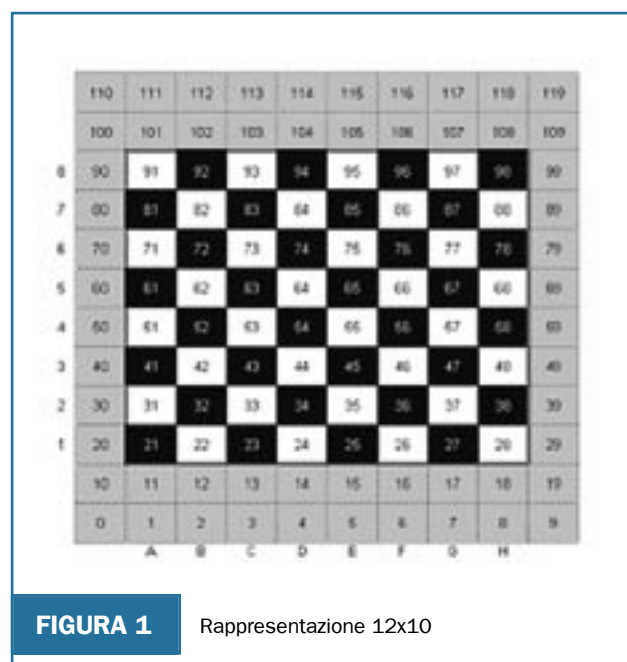
0 per le caselle al di fuori della scacchiera.

- 1 per il Pedone bianco
- 2 per il Pedone nero
- 3 per il Cavallo bianco
- 4 per il Cavallo nero

- 5 per il Re bianco
- 6 per il Re nero
- 7 per la Torre bianca
- 8 per la Torre nera
- 9 per la Regina bianca
- 10 per la Regina nera
- 11 per l'Alfiere bianco
- 12 per l'Alfiere nero

Perché questi valori?

Perché in questo modo con una semplice operazione si può



**FIGURA 1**

Rappresentazione 12x10

stabilire se un pezzo è bianco oppure è nero:  
 Il pezzo è bianco se "Scacchiera(x) Mod 2 <> 0".  
 Il pezzo è nero se "Scacchiera(x) Mod 2 = 0".

Una rappresentazione ancora più efficace ma anche più complessa da implementare è la rappresentazione tramite Bitboard.

In questo caso la nostra scacchiera è rappresentata da una serie di bitboard di 64 elementi, dove ogni elemento è rappresentato da un bit. Naturalmente una bitboard da sola non basta, ma ne occorrono parecchie (ad esempio una bitboard potrebbe rappresentare le Torri bianche, un'altra le caselle sotto attacco dalle pedine avversari e così via...). Dodici bitboard (una per ogni tipo di pezzo) sono sufficienti per memorizzare una posizione della scacchiera.

Questa rappresentazione a mio avviso rappresenta il futuro nella programmazione scacchistica in quanto sui moderni processori a 64 bit generare le mosse con questa rappresentazione richiederebbe pochissimi cicli di clock.

Supponiamo ad esempio di dover verificare se il Re Nero è sotto scacco della Regina Bianca.

Con una rappresentazione della scacchiera a singolo array noi dobbiamo:

- Trovare la posizione della Regina che richiede una scansione lineare dei 64 elementi del nostro array.
- Esaminare tutte le caselle delle 8 possibili direzioni fino a quando non troviamo il Re Nero oppure usciamo dai limiti della scacchiera.

Invece con una rappresentazione tramite bitboard noi dobbiamo:

- Prendere la bitboard con la posizione della Regina Bianca.
- Utilizzarla come un indice da passare all'array che contiene tutte le caselle attaccate dalla Regina bianca per ognuna delle 64 posizioni.
- Eseguire un AND logico tra questa bitboard e quella con la posizione del Re Nero.

Se il risultato è diverso da 0 allora sappiamo che la Regina Bianca tiene sotto scacco il Re Nero, il tutto con semplici e veloci istruzioni.

### La generazione delle mosse

L'approccio da me utilizzato per la generazione delle mosse è la cosiddetta "generazione completa".

Per una determinata posizione vengono generate tutte le mosse "legali", cioè valide per tutti i pezzi in gioco.

Unica eccezione, non viene subito fatto un controllo se muovendo un nostro pezzo poniamo il nostro Re "sotto scacco" che come sappiamo è una mossa non valida.

Questo controllo lo facciamo nel passaggio successivo quando elaboriamo e simuliamo le mosse del giocatore avversario.

In pratica funziona così: giochiamo la mossa e poi facciamo

una simulazione di quale mossa potrebbe fare l'avversario.

Se il nostro Re viene catturato allora la mossa da noi fatta non è legale, di conseguenza la scartiamo e passiamo alla mossa successiva.

Questa tecnica ci evita per ogni mossa di richiamare la funzione che verifica se abbiamo il Re sotto scacco, delegando il tutto nella fase di generazione delle mosse dell'avversario.

Una volta generate, tutte le mosse vengono ordinate. L'ordinamento delle mosse (come vedremo nel prossimo articolo), costituirà un aspetto fondamentale nell'algoritmo di ricerca.

Di seguito viene rappresentata la funzione di generazione delle mosse in pseudo-codice; oltre a generare tutte le mosse per i nostri pezzi in gioco, verifichiamo anche se catturiamo il Re avversario. In caso affermativo la funzione restituirà il valore "Vero", altrimenti restituirà "Falso":

```
Public Function GeneraMosseBianco () As Boolean
    For I = 1 To "Numero Pezzi Bianchi"
        For J = 1 To "Numero di raggi in quella posizione"
            For K = 1 To "Numero di caselle per ogni raggio"
                If Scacchiera(Casella) = "Nostropezzo" Then Exit For
                If Scacchiera(Casella) = "Pezzo avversario" Then
                    If Scacchiera(Casella) = "Re avversario" Then
                        GeneraMosseBianco = True
                    Exit Function
                End If
            <Memorizziamo la mossa>
            <Le assegniamo un valore per l'ordinamento,
            questo sarà molto importante per La funzione
            di ricerca che vedremo più avanti>
        End if
    Next K
    Next J
    Next I
    <Ordiniamo le mosse in base al valore assegnato>
End Function
```

### Le chiavi Zobrist

È spesso necessario confrontare più volte una determinata disposizione dei pezzi sulla scacchiera per stabilire ad esempio se quella posizione si è già verificata nel corso della partita.

Confrontare le posizioni di tutti i pezzi comporterebbe una forte penalizzazione della nostra funzione di ricerca perché in pratica queste posizioni devono essere confrontate migliaia e migliaia di volte.

Una soluzione molto efficiente a questo problema è data dalle "Chiavi Zobrist".

In pratica ad ogni differente posizione viene assegnata una chiave da 64 bit (questa sarebbe l'ideale, ma siccome il VB6 non è particolarmente tollerante per la gestione di numeri di queste dimensioni ho utilizzato per Matilde 2 chiavi da 32 bit) ed in questo modo risulterà molto più rapido confrontare queste chiavi in quanto utilizzeremo delle semplici operazioni logiche.

L'utilizzo di chiavi da 64 bit si rende necessario per evitare il problema delle collisioni, cioè una stessa chiave che può rappresentare due posizioni diverse e che comporterebbe un grave errore di valutazione per il nostro programma.

Utilizzeremo a questo scopo una matrice (MatriceHASH(Pezzo, Colore, Casella)) ed una chiave (ChiaveHASH) che saranno inizializzati all'avvio del nostro motore. Alla matrice assegneremo dei valori casuali che saranno validi per tutta la durata della nostra applicazione, la ChiaveHASH sarà invece azzerata ad ogni nuova partita. Per aggiornare la ChiaveHASH di una nuova posizione realizzeremo una scansione della scacchiera e per ogni pezzo eseguiremo un'operazione di XOR.

Supponiamo ad esempio che la Torre Bianca in A1 catturi il Pedone nero in A7 e di dover aggiornare la Chiave HASH. Utilizzeremo questo procedimento:

- Cancelliamo il Pedone nero dalla casella A7:  
ChiaveHASH = ChiaveHASH XOR MatriceHASH(Pedone, Nero, A7)
- Aggiungiamo la Torre bianca alla casella A7:  
ChiaveHASH=ChiaveHASHXORMatriceHASH(Torre, Bianco, A7)
- Cancelliamo la Torre bianca dalla casella A1  
ChiaveHASH=ChiaveHASHXORMatriceHASH(Torre, Bianco, A1)

## Il conteggio delle ripetizioni

Se, durante la partita, una posizione si ripete per tre volte con lo stesso colore che deve muovere, la partita è patta per la regola delle *ripetizioni* (Threefold Repetition).

Sarebbe un peccato se il nostro motore in vantaggio magari di una Torre sull'avversario dovesse trovarsi in una condizione in cui è costretto a ripetere sempre le stesse mosse, magari per una serie di scacchi consecutivi. In tal caso la patta può essere richiesta e noi pareggeremo una partita che poteva facilmente essere vinta.

Le posizioni possono essere ripetute sia durante la partita (mosse già giocate), sia durante la nostra funzione di ricerca di tutte le mosse possibili (mosse ancora non giocate ma che il nostro motore si appresta a tenere in considerazione).

Beh, queste mosse devono sicuramente essere intercettate e trattate nella maniera più appropriata.

Il punteggio da assegnare a queste mosse generalmente dovrebbe essere quello di parità, con valori intorno allo zero, ma nel caso che noi si sia in vantaggio il punteggio da assegnare dovrebbe essere negativo (la mossa in pratica viene scartata) oppure il contrario avviene se stiamo perdendo, allora in tal caso diamo un valore positivo (la mossa viene accettata, meglio un pareggio che una sconfitta).

Il metodo implementato nel mio motore utilizza le *Chiavi Zobrist* (viste nel precedente paragrafo) per intercettare posizioni già giocate o analizzate in precedenza.

Avremo a disposizione un array dove saranno memorizzate le posizioni precedenti e non faremo altro che scorrerlo per verificare se la posizione che stiamo analizzando è già stata ripetuta in precedenza.

Ecco la funzione:

```
Public Function CheckRepetitions() As Boolean
Dim I As Byte
'Analizziamo la lista con le posizioni precedenti
For I = 0 To 254
If HASHList(I) = -99 Then 'Siamo arrivati alla fine
CheckRepetitions = False
Exit Function
Else
'Se abbiamo trovato una corrispondenza...
If HASHList(I) = HASHKey Then
CheckRepetitions = True
Exit Function
End If
End If
Next I
End Function
```

## Conclusioni

In questo articolo abbiamo analizzato uno dei più importanti aspetti della programmazione scacchistica, come rappresentare la nostra scacchiera interna. E' molto importante pianificare bene questo aspetto prima di cominciare ad implementare il nostro motore; una scelta non propriamente corretta potrebbe comportare delle grosse penalità nell'efficacia del motore e sicuramente un domani ci si troverà costretti a riscrivere ex-novo la nostra applicazione.

Altro aspetto importante è quello di intercettare posizioni ripetute che potrebbero comportare una dichiarazione di patta da parte dell'avversario magari proprio mentre eravamo in largo vantaggio su di esso.

Matilde utilizza a tale proposito le *Chiavi Zobrist*, un metodo veloce ed efficace per memorizzare posizioni che vengono ripetute migliaia e migliaia di volte. Un altro possibile utilizzo delle *Chiavi Zobrist* è quello implementato nelle HASH Table (tabelle HASH), un metodo per velocizzare (e di parecchio) la nostra funzione di ricerca nell'albero delle mosse.

## Bibliografia

- [1] Bruce Moreland, "Programming Topics", <http://www.brucemo.com/compchess/programming/index.htm>
- [2] Francois Dominic Laramée, "Chess Programming", <http://www.gamedev.net/reference/list.asp?categoryid=18>

## 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.