Università degli Studi di Pisa

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI
Corso di Laurea in Scienze dell'Informazione

Machine Learning basato su analisi di database. Il caso del gioco degli Scacchi.

Candidato
BENEDETTO TONI

Relatore
Chiar.mo Prof.
PAOLO CIANCARINI

Controrelatore
Chiar.mo Prof.
FRANCESCO ROMANI

Sessione estiva
Anno Accademico 1995-96

Sommario

Capitolo 1. Introduzione **9**
1.1 Giochi e Intelligenza Artificiale ... 9
1.2 Conoscenza nei giocatori artificiali di Scacchi ... 10
1.3 Machine Learning ... 11
1.4 Basi di dati nel ML applicato ai giochi ... 12
1.5 Obiettivi della tesi ... 13
1.6 Struttura della tesi ... 13
1.7 Ringraziamenti ... 14

Capitolo 2. Programmi di gioco **15**
2.1 Giochi ad informazione completa ... 15
2.2 Analisi dell'albero di gioco ... 17
2.2.1 L'algoritmo MiniMax ... 18
2.2.2 Variante NegaMax ... 19
2.2.3 L'algoritmo AlphaBeta ... 20
2.2.4 Variante Aspiration Search ... 23
2.2.5 Variante Fail Soft ... 24
2.2.6 Principal Variation AlphaBeta ... 25
2.2.7 Principal Variation Search ... 26
2.3 Euristiche per guidare la scelta della mossa ... 28
2.3.1 Libro delle aperture ... 28
2.3.2 Tabella delle trasposizioni ... 28
2.3.3 Euristiche di ordinamento delle mosse ... 29
2.3.4 Iterative Deepening ... 31
2.4 La funzione di valutazione statica ... 32
2.5 GnuChess 4.0 ... 33
2.5.1 Scelta della mossa ... 33
2.5.2 Funzione di valutazione ... 34
2.6 Conoscenza nei programmi di gioco ... 35

Capitolo 3. Machine Learning applicato al dominio dei giochi 38
3.1 Apprendimento per memorizzazione 38
3.2 Reti Neurali 40
3.2.1 Reti neurali nel Backgammon 40
3.2.2 Reti neurali negli Scacchi 41
3.2.3 Reti neurali per l'apprendimento di qualsiasi gioco 42
3.3 Apprendimento Bayesiano 43
3.3.1 Apprendimento Bayesiano nell'ambito dei giochi 44
3.4 Algoritmi Genetici 45
3.4.1 Algoritmi genetici negli Scacchi 46
3.5 Regressione lineare 47
3.6 Riconoscimento di pattern 48
3.7 Approcci ibridi 49
3.7.1 Il progetto Morph 49
3.8 Considerazioni generali 51

Capitolo 4. Creazione automatica del libro di aperture 54
4.1 Introduzione 54
4.1.1 Libri senza gestione delle trasposizioni 55
4.1.2 Libri con gestione delle trasposizioni 56
4.2 Creazione di un libro di aperture 56
4.3 Codifica delle informazioni e scelte di progetto 57
4.3.1 Probabilità di errore nell'uso delle tabelle di apertura 58
4.4 Uso della tabella di apertura 59
4.5 Sperimentazione 63
4.5.1 Generazione delle tabelle di apertura 64
4.5.2 Condizioni dei test 66
4.5.3 Risultati 70

Capitolo 5. Euristiche di preordinamento delle mosse 73
5.1 Prima tecnica 73
5.1.1 Implementazione 74
5.1.2 Sperimentazione 75
5.1.3 Ulteriore miglioramento tramite feedback sul database 76
5.2 Seconda Tecnica 79
5.2.1 Implementazione 80
5.2.2 Sperimentazione 83

Capitolo 6. Applicazione di tecniche di ML per la funzione di valutazione statica 86
6.1 Apprendimento Bayesiano 86
6.1.1 Apprendimento Bayesiano nei giochi a informazione completa 87
6.1.2 Apprendimento Bayesiano in GnuChess 90
6.1.3 Sperimentazione 94
6.2 Algoritmi genetici 97
6.2.1 Algoritmi genetici nei giochi a informazione completa 97
6.2.2 Algoritmi genetici applicati a GnuChess 99
6.2.3 Sperimentazione 100

Capitolo 7. Conclusioni 103
7.1 Lavoro svolto 103
7.2 Prospettive future 105

Appendice A. GnuChess 4.0 107
A.1 Rappresentazione dello stato del gioco e scelta della mossa 107
A.2 Generatore di mosse 111
A.3 Tabella delle trasposizioni 113
A.4 Libro delle aperture 115
A.5 La funzione di valutazione 117
A.5.1 La funzione ExaminePosition 117
A.5.2 La funzione evaluate 118
A.5.3 La funzione ScorePosition 119
A.6 Test su file di posizioni 126

Appendice B. Deja Vu Chess Library 127
B.1 Formato del database 127
B.2 Alcuni dati statistici 129

Bibliografia 134

Lista delle figure

Figura 2.1. Primi livelli dell'albero di gioco degli Scacchi. 16
Figura 2.2. Un albero di gioco visitato con la strategia A. 18
Figura 2.3. Un albero di gioco visitato con la strategia B. 18
Figura 2.4. Tagli provocati dall'algoritmo AlphaBeta. 21
Figura 2.5. Esempio di mosse killer. Muove il Bianco. 30
Figura 3.1. Schema di base di un algoritmo genetico. 46
Figura 3.2. Esempio di codice genetico di un giocatore artificiale 46
Figura 4.1. Deep Blue - Fritz. Posizione dopo 12. Dh5. 56
Figura 4.2. Rappresentazione grafica e testuale dei dati nella tabella di apertura per le posizioni raggiungibili con una mossa del bianco nella posizione del diagramma 63
Figura 4.3. Diagramma del flusso dei dati per l'esecuzione dei test. 64
Figura 4.4. Grafico della funzione utilizzata per stabilire il risultato delle partite interrotte alla 50ª mossa. 67
Figura 4.5. Ciclo di base di GnuChess versione Server 68
Figura 4.6. Ciclo di base di GnuChess versione Client. 69
Figura 5.1. Grafico della funzione che determina il risultato della partita in base allo score assegnato dalla valutazione. 77
Figura 5.2. Andamento della percentuale di punti vinti da OPTA FB. 78
Figura 5.3. Diagramma del flusso dei dati con l'impiego delle *HistoryTable*. 80
Figura 5.4. Una posizione di partenza per costruire una *HistoryTable*. 82
Figura 6.1. Schema del calcolo della nuova funzione di valutazione statica 91
Figura 6.2. Una mossa debole di BayesPlayer. 94
Figura 6.3. Esempio di riproduzione. 99

Figura A.1. Schema della funzione SelectMove() 109
Figura A.2. Ciclo di base di GnuChess 4.0 110
Figura A.3. Porzione del file di testo contenente le varianti di apertura. 116
Figura A.4. Valore della variabile *stage* in diverse fasi della partita. 118
Figura A.5. Esempio di valutazione statica. a) L'Alfiere in g5 si merita il bonus PINVAL. b) In questo caso invece viene assegnato il bonus XRAY. 122
Figura A.6. Esempio di valutazione di una posizione 125
Figura A.7. Esempio di notazione Forsyth di una posizione di test. 126
Figura B.1. Relazione tra i record di dejavu.DBF e quelli di dejavu.FPT. 128
Figura B.2. Un record di Deja Vu. 129
Figura B.3. Distribuzione temporale delle partite in Deja Vu. 131
Figura B.4. Distribuzione temporale delle partite in Deja Vu dal 1950 in poi 131

Lista delle tabelle

Tabella 4.1. Dati statistici riguardanti la tabella d'apertura sulla Difesa Francese. 65
Tabella 4.2. Dati statistici riguardanti la tabella d'apertura sulla Difesa Est-Indiana. 65
Tabella 4.3. Dati statistici riguardanti una tabella d'apertura ricavata da tutte le 353.000 partite di Deja Vu (dati approssimativi stimati). 65
Tabella 4.4. Match di GnuChess 4.0 contro se stesso....70
Tabella 4.5. Risultati test sulla Difesa Francese. 71
Tabella 4.6. Risultati test sulla Difesa Est-Indiana. 71
Tabella 5.1. Test di OPTA 3.1 sull'apertura Francese....75
Tabella 5.2. Test di OPTA 3.1 sull'apertura Est-Indiana....75
Tabella 5.3. Confronto tra il numero di nodi visitati da GnuChess 4.0 e da OPTA 3.1....76
Tabella 5.4. Match di OPTA FB. 79
Tabella 5.5. Le mosse con punteggio più alto presenti nella *HistoryTable* relativa alla posizione del diagramma nella fig. 5.4....82
Tabella 5.6. Confronto tra il numero di nodi visitati da OPTA 3.1 e OPHISTA. 84
Tabella 5.7 Esperimenti con *HistoryTable*. 84
Tabella 6.1. Numero di operazioni per il calcolo di $f_1(x)$. 93
Tabella 6.2. Esperimenti con apprendimento Bayesiano. 96
Tabella 6.3. Esperimenti con algoritmo genetico. 101
Tabella 6.4. Match di Genetic 2 su altre aperture....102
Tabella 7.1. Esperimento conclusivo. 105
Tabella A.1. Bonus per il preordinamento delle mosse. 113
Tabelle B.1. Record del file dejavu.DB....129
Tabella B.2. Record del file dejavu.FPT....129

Tabella B.3. Statistica dei risultati delle partite in Dejavu raggruppate per aperture. 132
Tabella B.4. I giocatori di cui sono riportate più partite in Deja Vu. . 133

Capitolo 1

Introduzione

1.1 Giochi e Intelligenza Artificiale

I giochi costituiscono per l'Intelligenza Artificiale un campo di ricerca privilegiato, in quanto forniscono comodi modelli di problemi reali. I giochi infatti, pur presentando problemi di complessità paragonabile a quelli reali, hanno regole ben definite e formalizzabili. Inoltre in questo campo esistono esperti in grado di giudicare la bontà dei risultati proposti da una macchina. Risulta quindi comodo lavorare in un ambito ben definito, per poi generalizzare e adattare i risultati così ottenuti a problemi più "seri".
Tra i giochi che maggiormente sono stati studiati ci sono quelli ad informazione completa, cioè con due giocatori che a turno eseguono una mossa tra quelle consentite, e conoscono istante per istante lo stato completo del gioco. Giochi di questo tipo sono la Dama, il Go, Othello, ecc. L'attenzione dei ricercatori si è prevalentemente soffermata sugli Scacchi: basti pensare che i princìpi di base degli attuali giocatori artificiali furono enunciati da Shannon nel 1948 [Sha50], e già nel 1951 Turing applicò, manualmente data la scarsa potenza dei computer di allora, un algoritmo di sua invenzione in grado di giocare a Scacchi [Tur53]. La scelta degli Scacchi è, del resto, storicamente e geograficamente spiegabile: gli Scacchi sono sempre stati considerati il gioco d'intelligenza per eccellenza e sono diffusi da diversi secoli in occidente, mentre altri giochi, come il Go ad esempio, altrettanto complessi, sono ristretti in altre aree geografiche.

1.2 Conoscenza nei giocatori artificiali di Scacchi

Attualmente esistono una grande quantità di programmi che giocano a Scacchi (giocatori artificiali). Più o meno tutti applicano l'algoritmo AlphaBeta e sue varianti.
Per giocare bene a Scacchi, però, un programma deve avere anche conoscenza specifica dei princìpi strategici alla base del gioco. Schematicamente possiamo individuare questa conoscenza in quattro punti di applicazione distinti:

- *libro di apertura*: è un archivio di mosse o posizioni che aiuta il programma nella prima fase del gioco permettendogli di scegliere molto rapidamente la mossa quando si trova di fronte ad una posizione nota;
- *euristiche per la costruzione dell'albero di gioco*: aumentano l'efficienza dell'algoritmo AlphaBeta, che pur visitando un numero minore di nodi garantisce i medesimi risultati con alberi di gioco di uguale profondità. Quello che si ottiene è un risparmio di tempo di elaborazione a parità di profondità di ricerca;
- *funzione di valutazione*: è una delle componenti più delicate di un giocatore artificiale e ne determina in gran parte il suo valore complessivo (a questo proposito vedi [Sch86]). Questa funzione valuta una posizione (nodo foglia dell'albero di gioco) attraverso una serie di nozioni specifiche, tra loro combinate in un unico valore, che indica quanto una determinata posizione è buona per un giocatore.
- *database di finali*: sono archivi in cui sono registrate le mosse migliori per tutte le posizioni appartenenti ad alcuni tipi di finali con pochi pezzi sulla scacchiera, tipicamente da tre a cinque ([Cia92], pp. 216-223). Al contrario del libro di apertura, le mosse del database di finali sono state calcolate con una ricerca esaustiva. Perciò un programma di gioco che si trova in una posizione che è nel database di finali può giocare immediatamente la mossa suggerita senza paura di sbagliare.

Stabiliti questi punti il problema da risolvere, per chi sviluppa programmi di gioco, è come acquisire e successivamente trasferire questa conoscenza nel giocatore artificiale. La soluzione comunemente usata, a parte il caso del database di finali, è farsi aiutare da un giocatore umano esperto, e attraverso tentativi e intuito cercare di alzare il livello del gioco espresso dal programma. Questa metodologia comporta due inconvenienti principali:

- richiede un tempo considerevole;
- generalmente chi si occupa dello sviluppo del software ha difficoltà a ricavare conoscenze utili dagli esperti del gioco: un po' perché gli informatici usano un linguaggio e gli scacchisti ne usano un altro, e un po' perché i grandi maestri sono di solito restii a collaborare allo sviluppo di giocatori artificiali.

Alternative a questo metodo, in grado di superare queste difficoltà, sono date dalle tecniche di Machine Learning (ML).

1.3 Machine Learning

Seguendo la definizione di Simon diremo che il ML è:

> *...ogni cambiamento in un sistema che gli permette di migliorarsi nell'esecuzione dello stesso compito o di un altro dello stesso tipo.* [Sim83]

Da questa definizione si deduce che il ML rappresenta una forma di adattamento all'ambiente attraverso l'esperienza, analogamente a quanto avviene per gli organismi biologici o, in senso lato e in tempi più lunghi, per le specie viventi che vi si adattano geneticamente.
L'obiettivo del ML è far migliorare un sistema senza bisogno del continuo intervento umano. Per l'apprendimento sono necessarie due componenti:

- un insieme di dati relativi al dominio di applicazione;
- un algoritmo di apprendimento in grado di estrarre conoscenze utili dall'insieme dei dati.

Nel campo dei giochi, un algoritmo di apprendimento dà la possibilità ad un giocatore artificiale di incrementare il proprio livello di gioco attraverso un ampliamento o un raffinamento delle conoscenze specifiche che il sistema possiede riguardo il proprio dominio di applicazione.
È da sottolineare che la qualità dei dati su cui l'algoritmo lavora determina in gran parte l'esito di tutto il processo di apprendimento. Un algoritmo per quanto furbo ed efficiente non riuscirà mai ad estrarre conoscenza utile da un insieme di dati che non contiene informazioni rilevanti sul dominio di applicazione.
Nel caso degli Scacchi in genere si considera l'insieme dei dati costituito da partite, che possono avere diversa origine, come già indicato da Samuel [Sam59] per il gioco della Dama:

- partite giocate dal sistema che apprende contro se stesso;
- partite giocate dal sistema contro un allenatore;
- partite di forti giocatori.

In questa tesi verrà trattato in particolare quest'ultimo caso.

1.4 Basi di dati nel ML applicato ai giochi

La scelta di usare un database di partite di forti giocatori per l'apprendimento nei giocatori artificiali sembra abbastanza naturale. L'idea è di poter acquisire, dall'osservazione delle mosse eseguite dai maestri, le conoscenze che sono all'origine della scelta di queste mosse tra quelle possibili. Il problema da risolvere è come riuscire a individuare queste conoscenze che sono implicite nella base di dati. A questo scopo possono servire varie tecniche classiche di ML (alcune delle quali sono illustrate nel capitolo 3).
Usare basi di dati per il ML è conveniente sotto diversi punti di vista:

- abbrevia i tempi di apprendimento presentando al sistema esempi attendibili (partite di livello magistrale) piuttosto che mosse di scarso interesse se non addirittura forvianti;
- rende superfluo l'intervento di allenatori umani o artificiali durante la fase di apprendimento da parte del sistema.

L'inconveniente potrebbe essere che per creare dal nulla un grande database (dell'ordine delle centinaia di migliaia o milioni di partite di alto livello) occorrono secoli di tempo/uomo, comprendendo anche il tempo di studio necessario per arrivare a giocare bene oltre al tempo materiale necessario per giocare le partite. Questo sarebbe un ostacolo piuttosto serio per un gioco appena inventato, ma gli Scacchi sono talmente diffusi e studiati da avere ormai una vasta letteratura e un gran numero di giocatori di classe magistrale, tali da garantire più che adeguate fonti di informazioni per l'apprendimento. In questi ultimi anni, inoltre, sono disponibili in commercio molti database di partite, anche di notevoli dimensioni (si può superare il mezzo milione di partite), grazie anche alla diffusione di memorie di massa di elevata capacità su CD-ROM.

1.5 Obiettivi della tesi

Questa tesi si propone di esplorare la possibilità di migliorare un giocatore artificiale attraverso tecniche di ML, che a partire da un grande database di partite sintetizzano, sotto varia forma, conoscenze utili per il gioco artificiale.
In particolare verrà preso in considerazione il programma di pubblico dominio Gnuchess 4.0 (vedi appendice A), e un database di oltre 350.000 partite (vedi appendice B).
A partire dai dati presenti nel database verranno perseguiti i seguenti obiettivi:

- creare automaticamente un libro di aperture;
- sviluppare euristiche capaci di incrementare l'efficienza dell'algoritmo di visita dell'albero di gioco;
- migliorare l'efficienza della funzione di valutazione con tecniche di ML.

Nelle varie fasi del lavoro verranno create delle versioni modificate di GnuChess 4.0. Al fine di controllarne la qualità, queste verranno confrontate con la versione originale, attraverso match di 20 partite. La disputa dei match sulla distanza delle 20 partite è un buon compromesso tra tempo di elaborazione richiesto e attendibilità dei risultati ottenuti. Secondo studi statistici la distanza delle 20 partite ha un livello di confidenza, cioè la probabilità che il test riveli la reale classe di appartenenza dei giocatori, dell'89%. Per arrivare a un livello di confidenza del 95% occorrerebbero altre 10 partite, un costo sproporzionato rispetto alla maggiore precisione ottenuta (vedi [Elo86], p. 29). Inoltre questa distanza è già stata usata per diversi esperimenti su giocatori artificiali di Scacchi ([Sch86], [BEGC90], [Toz93], [San93]), e tra giocatori umani è stata adottata nel match per il titolo mondiale Pga svoltosi nel 1995 tra Kasparov e Anand.

1.6 Struttura della tesi

Nel capitolo 2 verrà illustrato lo schema di base dei programmi di gioco, in particolare verranno descritti i principali algoritmi di analisi di alberi di gioco e lo struttura generale della funzione di valutazione statica. Inoltre si vedrà la struttura di base di un particolare programma di gioco: GnuChess 4.0. Nel capitolo 3 saranno passate in rassegna varie tecniche di ML applicate al dominio dei giochi: di queste

tecniche saranno presentati brevemente i princìpi di base e i risultati delle loro implementazioni. Nel capitolo 4 verrà illustrato un metodo per la generazione automatica di un libro di aperture a partire da un database di partite. Nel capitolo 5 saranno illustrate due tecniche di preordinamento delle mosse capaci di migliorare le prestazioni dell'algoritmo AlphaBeta. Entrambe le tecniche faranno uso delle informazioni ricavate da database di partite. I giocatori artificiali ricavati dall'implementazione di queste tecniche in GnuChess 4.0 saranno confrontati con la versione originale di GnuChess 4.0 tramite match. Nel capitolo 6 saranno condotti esperimenti per il miglioramento dell'efficienza della funzione di valutazione statica. In particolare saranno usate due tecniche di ML: apprendimento Bayesiano e algoritmi genetici. Anche in questo caso, i giocatori artificiali risultanti dall'applicazione di queste tecniche saranno confrontati con la versione originale di GnuChess 4.0. Infine, nel capitolo 7 si trarranno le conclusioni sul lavoro svolto e si indicheranno le prospettive per eventuali lavori futuri.

1.7 Ringraziamenti

Si desidera ringraziare la Free Software Foundation per la concessione del programma GnuChess 4.0, Paolo Cecchetti per aver portato questo programma su piattaforma PowerPC, e il Prof. Francesco Romani per quanto riguarda l'uso del database di partite.

Capitolo 2

Programmi di gioco

2.1 Giochi ad informazione completa

I giochi ad informazione completa sono quei giochi in cui sia la posizione, sia tutte le decisioni possibili sono note a tutti i partecipanti in ogni momento. In questo capitolo in particolare ci occuperemo di giochi con due avversari che alternano le mosse e in cui vale la proprietà *0-sum* (ciò che un giocatore guadagna è equivalente a ciò che l'altro perde). A questa classe di giochi appartengono anche gli Scacchi.
Il modello matematico discreto con cui è possibile rappresentare i problemi decisionali posti da questo tipo di giochi è l'albero di gioco. Questa struttura è costituita da nodi, che rappresentano i possibili stati del gioco, e da archi, che rappresentano le mosse. Gli archi uscenti da un nodo corrispondono alle mosse legali nello stato del gioco associato a quel particolare nodo. La radice dell'albero è lo stato iniziale del gioco (vedi fig. 2.1). I nodi foglia dell'albero sono gli stati finali del gioco, in cui non esistono mosse legali, a questi nodi è associato un valore che indica il risultato della partita. Nei giochi molto semplici (per es. il Tic-tac-toe) l'albero di gioco ha una dimensione piuttosto ridotta, ma nei giochi più complessi assume dimensioni enormi (per es. l'albero di gioco degli Scacchi ha alcune migliaia di livelli e in media ogni nodo ha una trentina di figli).

a2a3
a2a4
Ng1h3
a7a6
a7a5
Ng8h6
Ng8h6

Figura 2.1. Primi livelli dell'albero di gioco degli Scacchi.

2.2 Analisi dell'albero di gioco

Per costruire un giocatore artificiale occorre una strategia che sia in grado di decidere, dato uno stato del gioco (quindi un nodo dell'albero di gioco), quale sia la migliore mossa da giocare. Prima di tutto occorre considerare che la grande dimensione dell'albero di gioco degli Scacchi rende impensabile una sua analisi esaustiva per decidere la mossa da giocare. Perciò l'analisi deve necessariamente essere limitata ad un albero di gioco parziale. I nodi foglia di questo albero parziale possono non essere stati finali del gioco e quindi non avere associata alcuna informazione sul risultato della partita. Una qualche valutazione di questi nodi è però necessaria per decidere la mossa da giocare. Perciò i nodi foglia dell'albero parziale vengono valutati da una apposita funzione, detta di valutazione statica, che cerca di approssimare il risultato finale della partita sulla base dello stato attuale del gioco.
Un albero parziale si può ottenere principalmente in due modi, secondo la classificazione di Shannon [Sha50]:

- *strategia A*: limitando l'estensione dell'albero, dal nodo attuale, ad una certa profondità (vedi fig. 2.2);

- *strategia B*: considerando non tutte le mosse legali, ma solo quelle che risultano più interessanti secondo dei criteri euristici (vedi fig. 2.3).

Gli attuali programmi di gioco adottano in genere la strategia A, detta anche di "espansione cieca", e la strategia B, o di "espansione euristica", solo in casi particolari (ad es. per la ricerca quiescente). L'insuccesso dei programmi che adottano solo la strategia B (per es. Pioneer [Bot82]) è dovuto sostanzialmente alla difficoltà di stabilire a priori quali siano le mosse interessanti e quali quelle da scartare.

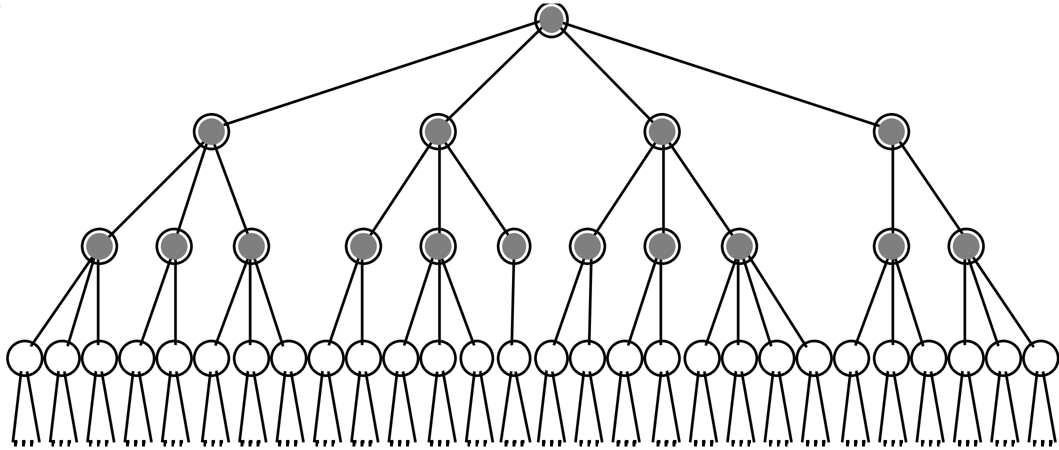


Figura 2.2. Un albero di gioco visitato con la strategia A fino al secondo livello di profondità. I nodi grigi sono quelli visitati.

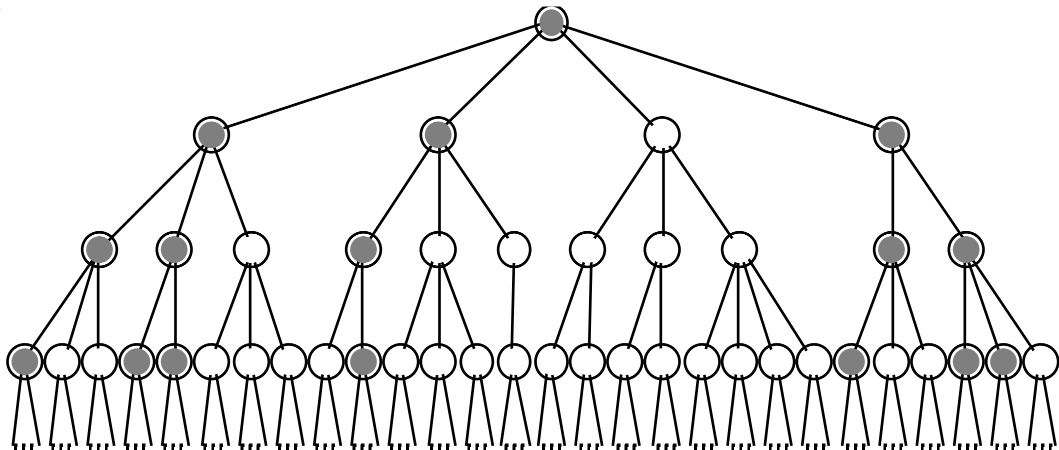


Figura 2.3. Un albero di gioco visitato con la strategia B. I nodi grigi sono quelli visitati.

2.2.1 L'algoritmo MiniMax

Partendo dall'albero di gioco in cui solo i nodi foglia sono valutati, l'algoritmo MiniMax esegue brutalmente una visita (di solito depth-first) di tutto l'albero, assegnando ai nodi non valutati il massimo o il minimo dei valori dei figli, rispettivamente a seconda che in quel nodo la mossa spetti al giocatore Max o al giocatore Min. Volendo quindi sce-

gliere la mossa da giocare in un determinato stato del gioco s, dopo l'applicazione dell'algoritmo MiniMax al sottoalbero che ha per radice s, basterà scegliere la mossa che porta al suo figlio che ha la valutazione più vantaggiosa dal punto di vista del giocatore che ha la mossa.
Più formalmente l'algoritmo può essere così descritto[1]:

```
// input : il nodo da valutare e il relativo sottoalbero di gioco
           e il giocatore che ha la mossa
// output: il valore del nodo dato

value MiniMax(nodo,gioc)
{
value val[MAXSONS];

      if (nodo è una foglia) return (valore(nodo));

      for (ogni figlio son[i] di nodo)
            val[i] = MiniMax (son[i], avversario(gioc));
      if (gioc == Max) return (max(val[]);
      return (min(val[]);
}
```

2.2.2 Variante NegaMax

La variante NegaMax differisce dall'algoritmo MiniMax per il modo in cui sono assegnati i valori ai nodi. Anziché assegnare un valore indipendente da chi ha la mossa in ciascuna posizione, si assegna un valore relativo al giocatore che deve muovere: lo 0 indica una situazione di parità, un valore positivo indica un vantaggio del giocatore che ha la mossa e uno negativo uno svantaggio. Quando i valori vengono propagati verso la radice dell'albero, non ci interessa più distinguere tra i giocatori Max e Min, basta semplicemente assegnare al nodo padre il massimo dei valori cambiati di segno dei figli.
La struttura dell'algoritmo è quindi la seguente:

```
// input : il nodo da valutare e il relativo sottoalbero di gioco
```

[1]Tutti i brani di codice sorgente, descrizioni di strutture dati, descrizioni di algoritmi, sono espressi nella sintassi del linguaggio C.

```
// output: il valore del nodo dato, relativo al giocatore che ha
           la mossa

value NegaMax(nodo)
{
value val[MAXSONS];

      if (nodo è una foglia) return (valore_per_chi_muove(nodo));

      for (ogni figlio son[i] di nodo)
            val[i] = - NegaMax (son[i]);
      return (max(val[]));
}
```

L'algoritmo NegaMax è equivalente al MiniMax. Infatti se chiamiamo *val*(*x,player*) e *val*(*x,otherplayer*) rispettivamente il valore del generico nodo *x* dal punto di vista di un giocatore e del suo avversario, per la proprietà *0-sum* si ha

$$val(x, player) = -val(x, otherplayer)$$

Sfruttando questa uguaglianza, riformuliamo i due diversi modi di propagare i valori nell'algoritmo MiniMax. Si ha:

per Max: $val(x, me) = \max_i(val(x_i, me)) = \max_i(-val(x_i, otherplayer))$
per Min: $val(x, me) = -\min_i(val(x_i, otherplayer)) = \max_i(-val(x_i, otherplayer))$

Come si vede le parti destre di queste uguaglianze sono identiche e corrispondono al metodo di propagazione dei valori utilizzato dall'algoritmo NegaMax.
Gli algoritmi che verranno descritti nel seguito di questo capitolo useranno tutti la modalità di propagazione dei valori usata nel NegaMax.

2.2.3 L'algoritmo AlphaBeta

L'algoritmo AlphaBeta nasce dall'osservazione che molti nodi visitati dall'algoritmo MiniMax (o NegaMax) non contribuiscono alla valutazione della radice del sottoalbero di gioco che viene analizzato. L'idea

di base dell'algoritmo è che quando si è scoperto che una variante non contribuirà alla valutazione del nodo radice, è inutile continuare ad analizzarla.
Esempi di tagli dell'albero di gioco sono riportati nella figura 2.4. La visita dell'albero avviene in ordine depth-first. I valori dai nodi foglia alla radice vengono propagati con la modalità dell'algoritmo NegaMax. La visita del primo ramo della radice porta a una valutazione provvisoria pari a -1 dovuta alla propagazione del valore del nodo A. Nella visita del nodo B si può osservare che il valore -4 del primo figlio rende superflua la visita degli altri figli: qualsiasi sia il valore di questi figli, il nodo B avrà una valutazione ≥4, ma dato che la radice ha già una valutazione provvisoria pari a -1, il nodo B non potrà mai essere scelto come continuazione principale. Le stesse osservazioni valgono anche per i tagli nei nodi C e D.
Per controllare quando tagli come quelli in figura 2.4 sono possibili, sono introdotti due valori, alpha e beta, che vengono propagati di padre in figlio durante la visita dell'albero di gioco. Questi valori vengono ricalcolati durante la visita in modo che ciascun nodo contribuisca a determinare il nuovo valore della radice solo se la propria valutazione è compresa tra alpha e beta correnti.

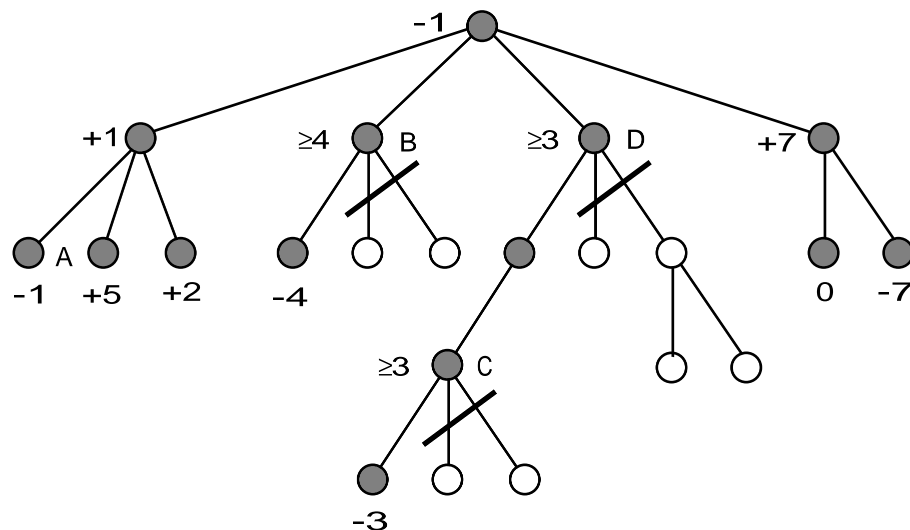


Figura 2.4. Tagli provocati dall'algoritmo AlphaBeta. I nodi effettivamente visitati sono solo quelli in grigio.

Formalmente l'algoritmo AlphaBeta versione NegaMax può essere così descritto:

```
// input : il nodo da valutare e il relativo sottoalbero di gioco
           e gli estremi della finestra alpha-beta
// output: il valore nel nodo dato, relativamente al giocatore che
           ha la mossa

value AlphaBeta(nodo,alpha,beta)
{
value val[MAXSONS]

      if (nodo è una foglia) return (valore_per_chi_muove(nodo));

      for (ogni figlio son[i] di nodo)
      {
            val[i] = - AlphaBeta (son[i], -beta, -alpha);
            if (val[i]>alpha)
                  alpha = val[i];     // nuova mossa migliore
            if (val[i]>beta)
                  return (+INFINITO); // taglio
      }

      return(alpha);
}
```

La prima chiamata della funzione `AlphaBeta` all'inizio della ricerca della mossa da giocare ha per parametri la radice dell'albero di gioco da visitare e i valori $\alpha = -\infty$, $\beta = +\infty$.
Le prestazioni dell'algoritmo AlphaBeta sono fortemente legate all'ordine in cui i nodi vengono visitati, infatti i tagli avvengono solo se una variante migliore di quella che si sta analizzando è già stata trovata, altrimenti il numero di nodi visitati dall'algoritmo non varia rispetto al normale MiniMax. Il caso ottimo si verifica quando i figli di un nodo vengono visitati in ordine decrescente: è stato calcolato che i nodi foglia visitati, considerando un albero uniforme (con B figli per ogni nodo) di profondità D, siano $2B^{D/2} - 1$ se D e pari, e $B^{(D-1)/2} + B^{(D+1)/2} - 1$ se D è dispari [Gil78], mentre nel caso che i nodi non siano ordinati si stima che i nodi foglia siano dell'ordine di $B^{D/2}\log_2 B$ [Lev91], contro i B^D nodi foglia visitati dal MiniMax. Quindi si può dire che a parità di

tempo i programmi che usano l'algoritmo AlphaBeta possono raggiungere una profondità di analisi quasi doppia rispetto ai programmi che adottano il semplice MiniMax.

2.2.4 Variante Aspiration Search

La variante Aspiration Search differisce dall'algoritmo AlphaBeta per la finestra alpha-beta attribuita al nodo radice della ricerca. Anziché essere la canonica $\alpha = -\infty, \beta = +\infty$, la finestra iniziale viene calcolata sulla base di una valutazione provvisoria v del nodo radice e una costante K. Per cui si ha $\alpha = v - K$, e $\beta = v + K$. Lo scopo di questa nuova finestra è di generare un maggior numero di tagli. Tuttavia, se il valore reale della radice non è compreso nella finestra iniziale, è probabile che alcuni tagli effettuati non siano corretti, cioè che si scartino mosse che possono essere rilevanti per la valutazione della radice, in questo caso occorre ripetere la ricerca con una finestra alpha-beta che comprenda sicuramente il reale valore della radice. In particolare bisognerà porre $\alpha = -\infty, \beta = v - K$, se si è verificato un fallimento inferiore, oppure $\alpha = v + K, \beta = +\infty$, se si è verificato un fallimento superiore.
L'efficienza di questo algoritmo è fortemente legata alla capacità di fare una buona previsione del valore della radice, e alla scelta della costante K, che deve essere un compromesso tra il numero di tagli che si vogliono ottenere e la probabilità di dover ripetere la ricerca.
Lo schema dell'algoritmo è il seguente:

```
// input : il nodo da valutare e il relativo sottoalbero di gioco
// output: il valore nel nodo dato, relativamente al giocatore che
           ha la mossa

value AspirationSearch(nodo)
{
value v, firstVal, alpha, beta;

      v = stima_per_chi_muove(nodo);
      alpha = v - K;
      beta  = v + K;

      firstVal = AlphaBeta (nodo, alpha, beta);
```

```
      if (fistVal > beta) // fallimento superiore
            return (AlphaBeta (nodo, beta, +INFINITO));
      if (fistVal < alpha) // fallimento inferiore
            return (AlphaBeta (nodo, -INFINITO, alpha));
      return (firstVal);
}
```

2.2.5 *Variante Fail Soft*

Per migliorare le prestazioni dell'algoritmo Aspiration Search in caso di fallimento della prima ricerca è stato introdotta la variante Fail Soft. Questa variante consente di effettuare l'eventuale seconda ricerca con una finestra più piccola di quella usata dall'Aspiration Search, grazie alla seguente riformulazione dell'algoritmo AlphaBeta:

```
// input : il nodo da valutare e il relativo sottoalbero di gioco
            e gli estremi della finestra alpha-beta
// output: il valore nel nodo dato, relativamente al giocatore che
           ha la mossa

value F_AlphaBeta(nodo,alpha,beta)
{
value val[MAXSONS], best;

      if (nodo è una foglia) return (valore_per_chi_muove(nodo));

      best = -INFINITO;
      for (ogni figlio son[i] di nodo)
      {
            val[i] = - F_AlphaBeta(son[i],-beta,-max(alpha,best));
            if (val[i]>best)
                  best = val[i];     // nuova mossa migliore
                  if (val[i]>beta)
                        return (val[i]);    // taglio
      }

      return(best);
}
```

In questo modo il valore v restituito dalla funzione F_AlphaBeta(*nodo*, α, β) è il valore corretto di *nodo* se $v \in [\alpha, \beta]$, è un limite inferiore se $v > \beta$, è un limite superiore se $v < \alpha$. Utilizzando questa nuova funzione la versione Fail Soft dell'algoritmo Aspiration Search è:

```
// input : il nodo da valutare e il relativo sottoalbero di gioco
// output: il valore nel nodo dato, relativamente al giocatore che
           ha la mossa

value F_AspirationSearch(nodo)
{
value v, firstVal, alpha, beta;

      v = stima_per_chi_muove(nodo);
      alpha = v - K;
      beta  = v + K;

      firstVal = F_AlphaBeta (nodo, alpha, beta);

      if (fistVal > beta) // fallimento superiore
            return (AlphaBeta (nodo, firstVal, +INFINITO));
      if (fistVal < alpha) // fallimento inferiore
            return (AlphaBeta (nodo, -INFINITO, firstVal));
      return (firstVal);
}
```

2.2.6 *Principal Variation AlphaBeta*

Sfruttando i limiti forniti dalla funzione F_AlphaBeta è possibile costruire un algoritmo di ricerca che usa inizialmente una finestra alpha-beta minima (contenente un solo valore) e che riesegue la ricerca solo nel caso che la maggiorazione ottenuta con la prima ricerca non sia sufficiente per scartare la variante in esame. L'algoritmo Principal Variation AlphaBeta sfrutta questa idea: visitando un nodo della variante principale viene prima valutato esattamente il primo figlio (anch'esso nella variante principale); in base a questa valutazione viene costruita la finestra alpha-beta minima con cui vengono analizzati gli altri figli. Se la ricerca con la finestra minima restituisce una maggiorazione inferiore al valore del miglior figlio finora visitato allora si può

passare alla mossa successiva, altrimenti occorre stabilire il vero valore AlphaBeta con una ricerca su una finestra più ampia.
Formalmente l'algoritmo può essere così descritto:

```
// input : il nodo da valutare e il relativo sottoalbero di gioco
// output: il valore nel nodo dato, relativamente al giocatore che
           ha la mossa

value P_V_AlphaBeta(nodo)
{
value val[MAXSONS], best;

      if (nodo è una foglia) return (valore_per_chi_muove(nodo));

      best = - P_V_AlphaBeta(son[0]);
      for (ogni figlio son[i] di nodo, i>0)
      {
            val[i] = - F_AlphaBeta(son[i], -best-1, -best);
            if (val[i]>best)  // nuova mossa migliore
                  best= - F_AlphaBeta(son[i], -INFINITO, -val[i]);
      }
      return(best);
}
```

L'efficienza dell'algoritmo, anche in questo caso, è legata al fatto di visitare per primo il figlio corrispondente alla mossa migliore, in modo che per le restanti mosse sia sufficiente una rapida ricerca con finestra alpha-beta minima. Quando invece è necessaria una seconda ricerca le prestazioni dell'algoritmo possono essere anche peggiori della variante Fail Soft [Kai90].

2.2.7 Principal Variation Search

L'algoritmo Principal Variation Search è analogo al precedente, però la ricerca con finestra minima viene estesa, oltre alle alternative della variante principale, a tutte le varianti:

```
// input : il nodo da valutare e il relativo sottoalbero di gioco
           espanso fino alla profondità depth dalla radice, e
           gli estremi della finestra alpha-beta
// output: il valore nel nodo dato, relativamente al giocatore che
           ha la mossa

value PVS(nodo,alpha,beta,depth)
{
value val[MAXSONS], best;

      if (depth==0) return (valore_per_chi_muove(nodo));

      best = - PVS(son[0], -beta, -alpha, depth-1);
      if (best => beta) return (best); // taglio

      for (ogni figlio son[i] di nodo, i>0)
      {
            if (best > alpha) alpha = best;
            val[i] = - PVS(son[i], -alpha-1, -alpha, depth-1);
            if (val[i]>alpha and val[i]<beta)
                  // nuova mossa migliore
                  best = - PVS(son[i], -beta, -val[i], depth-1);
            else
                  if (val[i]>best)  // nuova mossa migliore
                        best=val[i];
            if (best => beta) return (best); // taglio
      }
      return(best);
}
```

In questo paragrafo sono stati presentati alcuni tra i più significativi algoritmi di ricerca della mossa per programmi di gioco che adottano la strategia di Shannon *tipo A*: dai più semplici MiniMax e NegaMax, qui presentati perché alla base degli sviluppi successivi, alle sofisticate varianti dell'algoritmo AlphaBeta. Di questi ultimi si è vista la forte dipendenza delle loro prestazioni dall'ordine di visita dei nodi. È quindi interessante studiare strategie che aiutino l'algoritmo a scegliere un ordinamento dei nodi il più possibile vicino all'ordinamento ottimo. Esempi di queste strategie sono riportate nel paragrafo 2.3.3. Anche nel

capitolo 5 saranno presentate strategie di questo tipo, che fanno uso di dati ricavati da database di partite, applicate all'algoritmo Aspiration Search variante Fail Soft, che è l'algoritmo di ricerca adottato da GnuChess.

2.3 Euristiche per guidare la scelta della mossa

Anche con gli algoritmi di analisi più sofisticati, la visita dell'albero di gioco è un'operazione dispendiosa, di complessità esponenziale rispetto alla profondità di ricerca, per cui diverse euristiche sono state introdotte per cercare di ottimizzarla.

2.3.1 Libro delle aperture

Il libro delle aperture è un archivio in cui a delle posizioni sono associate mosse da giocare. Quando il programma si trova a dover muovere in una posizione presente nel libro non viene usato l'algoritmo di analisi dell'albero di gioco, ma viene direttamente giocata una delle mosse presenti nel libro di apertura. L'uso del libro di aperture permette di risparmiare tempo nella scelta della mossa nella prima fase della partita, e se le mosse del libro sono buone, evita l'esecuzione di possibili mosse deboli da parte del programma.

2.3.2 Tabella delle trasposizioni

La tabella delle trasposizioni, utilizzata ormai da tutti i programmi di gioco, è una tabella hash in cui vengono memorizzate informazioni riguardanti le posizioni analizzate durante la ricerca. Queste informazioni possono poi essere utilizzate quando viene analizzata una posizione già incontrata, cioè quando si è verificata una trasposizione.
Le informazioni contenute nella tabella per ogni posizione sono in genere:

- un identificatore di posizione;
- una valutazione;
- la profondità di analisi con cui è stata determinata la valutazione;

- una indicazione se la valutazione è esatta oppure se è una limitazione superiore o inferiore (necessaria con algoritmi tipo Fail Soft).

La lettura di una entry della tabella può sostituire l'analisi di un intero sottoalbero di gioco quando la posizione alla radice del sottoalbero è nella tabella e la profondità di analisi con cui è stata determinata la valutazione nella tabella è maggiore o uguale a quella del sottoalbero in esame.
Data la limitata dimensione della tabella, normalmente piccola rispetto al numero di posizioni incontrate durante la ricerca, in essa non vengono inserite le posizioni valutate con una profondità di analisi inferiore ad una certa soglia stabilita dal programmatore. Nell'eventualità di conflitti possono essere usati diversi criteri di rimpiazzamento, ad esempio tenere in tabella la posizione con profondità di analisi maggiore, oppure dare la precedenza sempre all'ultima posizione trovata, ecc. (vedi [BUV94]).

2.3.3 Euristiche di ordinamento delle mosse

L'ordine in cui le mosse vengono analizzate dall'algoritmo AlphaBeta (o da sue varianti) è fondamentale ai fini della sua efficienza. Per questo sono state sviluppate diverse euristiche:

- *Ordinamento a priori* (denominato *Knowledge Heuristic* in [Sch86]): le mosse vengono ordinate, in mancanza di altre informazioni utili, in base a certe loro caratteristiche, per esempio vengono privilegiate le mosse di promozione o le mosse di cattura secondo il valore del pezzo catturato, ecc. Le caratteristiche che questo ordinamento prende in considerazione sono puramente di tipo scacchistico e non sono legate a informazioni ricavate in precedenza dall'algoritmo di ricerca, come nel caso delle euristiche che seguono.

- *Variante principale* [AklNew77]: questa euristica dà la precedenza alla mossa calcolata come migliore in una precedente visita della posizione che si sta analizzando. Questa euristica è utile quando la posizione in esame è nella tabella delle trasposizioni, ma lo score associato alla posizione non può essere preso come valutazione. Si prende allora la mossa associata alla entry nella tabella delle

trasposizioni solo come suggerimento per l'ulteriore ricerca che verrà condotta.

- *Mosse killer* [AklNew77], [Sch86]: sono considerate killer quelle mosse che provocano tagli durante la ricerca AlphaBeta. Capita spesso che una mossa killer in una variante, lo sia anche in altre varianti in cui compare alla stessa profondità. È questo il caso in cui la mossa killer porta delle minacce che molte mosse dell'avversario non riescono a sventare.
  Per questi motivi è vantaggioso, quando è possibile, dare la precedenza a queste mosse killer rispetto alle altre. In genere viene memorizzata una piccola lista di mosse killer per ogni livello dell'albero di gioco da analizzare.
  Un esempio di mossa killer è illustrato nella figura 2.5.

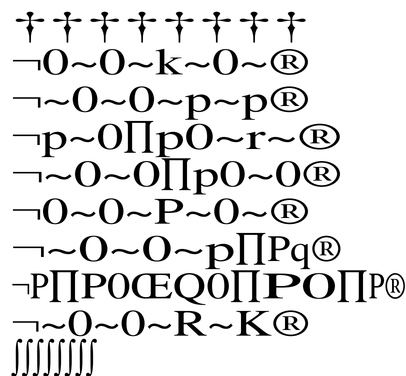


Figura 2.5. Esempio di mosse killer. Muove il Bianco.

Nella posizione del diagramma la mossa del Nero Dg2# è una mossa killer, infatti tutte le mosse del Bianco, esclusa Tg1, vengono confutate da questa mossa.

- *Tabelle History* [Sch83]: questa euristica è simile a quella delle mosse killer, in questo caso però non si tiene conto del livello di profondità in cui le mosse sono state giocate. Ogni volta che una mossa genera un taglio viene incrementato il proprio valore nella tabella history. Per ogni posizione, al momento di generare le mosse legali per proseguire l'analisi, le mosse vengono ordinate in base al numero di tagli che hanno già provocato in altri rami dell'albero di gioco già visitati, oppure in iterazioni precedenti dell'Iterative Deepening (vedi paragrafo successivo).

Queste euristiche in genere non compaiono mai da sole in un programma di gioco, ma sono tra loro combinate secondo criteri che variano da programma a programma. Un esempio di combinazione di diverse euristiche di ordinamento si può trovare in [Sch86]. Comunque di solito si dà priorità massima alla variante principale e minima all'ordinamento a priori, mentre un peso intermedio viene assegnato alle altre euristiche.

2.3.4 Iterative Deepening

L'Iterative Deepening (approfondimento iterativo, vedi [Kai90], p.144) è una modalità di ricerca in cui la visita dell'albero di gioco viene effettuata più volte, con un algoritmo di tipo AlphaBeta, a profondità crescente. Ad ogni iterazione le informazioni ricavate dall'iterazione precedente (variante principale, tabella delle trasposizioni, mosse killer ecc.) vengono sfruttate per velocizzare l'analisi, per cui l'apparente spreco di tempo dovuto al fatto di eseguire più volte la ricerca viene più che compensato dalle euristiche di velocizzazione che possono essere messe in atto. Un altro vantaggio di questa modalità di ricerca, indipendentemente dalla sua velocità, è la facilità con cui può essere combinata con un sistema di controllo di tempo, necessario per le normali partite di torneo: quando il tempo scade posso interrompere la ricerca in corso e giocare la mossa migliore trovata durante l'ultima iterazione che si è conclusa, oppure giocare una mossa, migliore di questa, che è stata trovata nell'iterazione che è stata interrotta.
Schematicamente questa modalità di ricerca può essere così descritta:

```
// input : il nodo da valutare e il relativo sottoalbero di gioco
// output: la mossa da giocare

move I_D_Search(nodo)
{
value d=0; move mv;

      while (il tempo non è terminato) and (d++ < MAXDEPTH) do
            mv = search(nodo,d);

      return(mv);
}
```

2.4 La funzione di valutazione statica

In un giocatore artificiale la funzione di valutazione statica rappresenta una delle parti fondamentali e più complesse. La qualità di un programma di gioco è in buona parte determinata dalla capacità della propria funzione di valutazione di stimare con precisione le posizioni nei nodi foglia dell'albero di gioco. Questa valutazione viene fatta attraverso l'osservazione di caratteristiche notevoli della posizione in esame. La posizione viene esaminata sotto diversi aspetti tramite l'uso di diversi *termini di conoscenza* (valore materiale dei pezzi, struttura pedonale, controllo del centro ecc.) che poi vengono sintetizzate in un unico valore tramite un'apposita funzione. Tipicamente una funzione di valutazione ha la seguente forma:

$$f(p) = \sum_{i=1}^{N} weight_i \cdot A_i(p)$$

dove con $A_i(p)$ si indicano i valori dei termini di conoscenza applicati alla posizione p e con $weight_i$ il peso relativo di ciascun termine di conoscenza.
Una importante caratteristica che una buona funzione di valutazione deve avere è di non essere rigida, cioè di operare in maniera diversa a seconda della posizione a cui è applicata. Per esempio è bene che la funzione di valutazione adotti termini di conoscenza o pesi diversi per le varie fasi della partita (tipicamente divisa in apertura, mediogioco e finale) in cui è noto che sono necessarie metastrategie e quindi conoscenze diverse. È anche utile che il giocatore artificiale sappia riconoscere situazioni particolari in cui sono necessarie conoscenze molto specifiche: per esempio in un finale di Re, Cavallo, Alfiere contro Re è inutile applicare una funzione di valutazione come quella per la fase di mediogioco, è necessario piuttosto applicare un insieme di regole strategiche appositamente studiate per questo tipo di finale.
Un problema ricorrente per la funzione di valutazione, chiamato "effetto orizzonte" (vedi [Kai90], p. 153), è valutare correttamente le posizioni in cui sono possibili molte mosse di cattura, promozione o scacco, chiamate posizioni turbolente. In queste posizioni è difficile dare una valutazione statica a causa dei numerosi tatticismi che vi sono e che possono essere visti solo analizzando ulteriormente l'albero di gioco. Per questo motivo di solito i programmi che adottano una strategia di espansione cieca fino ad una certa profondità usano una

tecnica di ricerca, chiamata ricerca quiescente, che permette di espandere ulteriormente l'albero di gioco per le posizioni turbolente considerando solo le mosse di cattura, scacco e promozione. La funzione di valutazione viene poi applicata ai nodi foglia dell'albero di gioco così espanso, che non contengono posizioni turbolente.

2.5 GnuChess 4.0

GnuChess 4.0 è un programma di Scacchi della Free Software Foundation, che mette a disposizione di tutti i sorgenti scritti in linguaggio C. Il programma viene sviluppato principalmente per il sistema operativo Unix, ma tramite opportune opzioni di compilazione è stato portato anche su MS-DOS e Macintosh. In particolare per questo lavoro è stata usata una versione compilata per Macintosh con processore PowerPC 601 e compilatore Metrowerks CodeWarrior.
Dal punto di vista agonistico GnuChess 4.0 possiede una forza paragonabile a quella di un maestro. Nei confronti degli altri giocatori artificiali sono da notare le vittorie nel torneo annuale su piattaforma stabile nel 1992 e 1993 [Bea92] [Bea93]. Comunque le sue prestazioni sono inferiori a quelle dei più forti programmi di gioco in commercio, ormai in grado di lottare ad armi pari con grandi maestri, se non addirittura con il campione del mondo, soprattutto nelle partite brevi.
GnuChess ha una struttura simile alla maggior parte dei programmi di gioco. Qui sono illustrate le caratteristiche principali della struttura interna del programma, per maggiori dettagli si rimanda alla appendice A.

2.5.1 Scelta della mossa

Al momento di scegliere la mossa da giocare, se non c'è un'alternativa suggerita dal libro delle aperture, viene eseguito un algoritmo di tipo Aspiration Search, versione Fail Soft, secondo la modalità dell'Iterative Deepening. La funzione di valutazione è chiamata per valutare tutti i nodi dell'albero di gioco. Durante la ricerca vengono visitati prima i nodi più promettenti, determinati in base a delle apposite euristiche. Inoltre in GnuChess è implementata una ricerca quiescente: l'albero di gioco viene espanso fino ad 11 semimosse oltre la profondità nominale limitatamente alle mosse di cattura, scacco e promozione, in

modo da procrastinare il più possibile il noto “effetto orizzonte”. È presente anche una tabella delle trasposizioni e una tabella delle trasposizioni permanente (non utilizzata per questa tesi, vedi paragrafo 3.1).

2.5.2 Funzione di valutazione

Alla funzione di valutazione è dedicata buona parte del codice sorgente. La sua struttura è fatta in modo che si comporti in maniera completamente diversa a seconda che la posizione in esame corrisponda ad una situazione particolare (tutti i finali in cui una parte è rimasta con il solo Re e l'avversario non ha pezzi, oppure non ha Pedoni) oppure no. Nel primo caso il compito di valutare la posizione viene demandato ad apposite procedure specializzate, mentre nel secondo caso, in cui ricadono la maggior parte delle posizioni, viene trattato con la funzione di valutazione vera e propria. Le nozioni scacchistiche che permettono a GnuChess di valutare una posizione possono essere suddivise in otto categorie di conoscenza:

1. Materiale (m): è la nozione strategica basilare per qualsiasi giocatore di Scacchi, a ciascun pezzo si attribuisce un valore, a prescindere dalla sua posizione sulla scacchiera. Il valore materiale del Re è convenzionalmente superiore a quello di tutti gli altri pezzi, in quanto perdere il Re significa perdere la partita. Per gli altri è universalmente riconosciuto che la Donna è il pezzo più forte, vale all’incirca come due Torri, mentre Alfiere e Cavallo hanno pressappoco lo stesso valore, pari ciascuno circa a tre Pedoni. Ogni Torre viene valutata all’incirca come cinque Pedoni.

2. Sistemazione dei pezzi (b): questo fattore di conoscenza premia i pezzi sistemati in case in cui non possono essere facilmente attaccati dagli avversari.

3. Spazio e mobilità (x): tra i due contendenti è di regola avvantaggiato chi controlla con i propri pezzi più case dell’avversario.

4. Sicurezza del Re (k): questo fattore permette di valutare quanto il proprio Re corre il rischio di subire un attacco da parte dei pezzi avversari.

5. Controllo del centro (c): è nozione comune di tutti gli scacchisti come controllare le case centrali favorisca le operazioni di attacco. Il concetto di controllo del centro è molto importante in fase di apertura, quando è l'obiettivo principale, non essendocene altri raggiungibili in poche mosse.

6. Struttura pedonale (p): sono note nella letteratura scacchistica strutture pedonali forti e strutture pedonali deboli (per es. con pedoni doppiati o isolati). Una struttura pedonale debole di solito rende debole tutta la posizione di un giocatore, soprattutto quando si arriva nel finale di partita, dove i pedoni acquistano una particolare importanza, dato che pochi pezzi sono rimasti sulla scacchiera.

7. Possibilità di attacco (a): questo fattore valuta le possibilità di attacco contro obiettivi concreti come il Re o debolezze della posizione avversaria, oppure la possibilità di portare un Pedone a promozione.

8. Relazione tra i pezzi (r): alcune combinazioni di pezzi dello stesso colore vengono preferite rispetto ad altre (un es. tipico è la coppia degli Alfieri, ritenuta in genere superiore alla coppia Cavallo-Alfiere).

2.6 Conoscenza nei programmi di gioco

Come si può notare da questa breve illustrazione della struttura dei giocatori artificiali, ci sono diversi tipi di conoscenze sintetizzate nei programmi di gioco. Schematicamente le possiamo dividere in due gruppi:

- conoscenze generali sui giochi a informazione completa;

- conoscenze specifiche del gioco di applicazione.

Delle conoscenze del primo gruppo possiamo considerare i vari algoritmi di ricerca nell'albero di gioco e alcune euristiche di velocizzazione della ricerca stessa: tabella delle trasposizioni, mosse killer, tabelle history, tabella delle confutazioni, iterative deepening. Queste conoscenze sono indipendenti dal particolare gioco a cui vengono applicate, e infatti vengono usate per giochi diversi come l'Othello, la Dama, gli Scacchi, ecc.

Nelle conoscenze specifiche del gioco di applicazione possiamo riconoscere, oltre alle regole del gioco, necessarie per sviluppare l'albero di gioco:

- il *libro delle aperture*: racchiude le conoscenze accumulate nello studio plurisecolare condotto sulla fase di apertura del gioco degli Scacchi;

- *euristiche di ordinamento a priori delle mosse durante l'analisi*: come abbiamo visto per ordinare le mosse, a prescindere dalle euristiche delle mosse killer, tabelle history, ecc., occorre conoscenza specifica, si tratta infatti di decidere quali siano le mosse più promettenti in una determinata posizione basandosi soltanto su considerazione puramente scacchistiche;

- la *funzione di valutazione statica*: in questa funzione è racchiusa gran parte della conoscenza del giocatore artificiale sul gioco degli Scacchi. È la funzione di valutazione che gli consente di decidere quale mossa scegliere tra le varie possibilità. In questa funzione sono sintetizzate conoscenze ricavate dallo studio del gioco che permettono di individuare in ogni posizione le caratteristiche principali per stabilire in che misura una posizione è migliore di un'altra dal punto di vista di uno dei due giocatori.

La fondamentale importanza delle conoscenze specifiche, specialmente di quelle indicate negli ultimi due punti, è messa bene in evidenza dagli studi di Schaeffer [Sch86] sul suo programma Phoenix. I risultati di questi esperimenti hanno mostrato la stretta relazione tra quantità di conoscenze inserite nella funzione di valutazione e qualità di gioco del programma. Inoltre, gli esperimenti di Schaeffer hanno dimostrato che usare conoscenza specifica per ordinare i nodi intermedi dell'albero di gioco può provocare tagli significati e quindi consentire notevoli risparmi di tempo di riflessione.
Tuttavia le conoscenze specifiche richiedono uno studio particolarmente approfondito del gioco di applicazione per essere individuate. La difficoltà, poi, di inserire queste conoscenze in un giocatore artificiale, fanno sì che non sia stato trovato un insieme ottimo di conoscenze.
Per esempio, la funzione di valutazione può essere definita in una infinità di modi: considerando insiemi di caratteristiche diversi della posizione e dando pesi relativi diversi alle varie caratteristiche, oppure

si possono considerare un maggior numero di termini di conoscenza, a discapito della velocità di analisi, oppure se ne possono considerare meno per avere una maggiore profondità di ricerca. In assoluto non si può dire quale soluzione per la funzione di valutazione sia la migliore, solo il confronto diretto, o test su posizioni campione, può stabile tra diverse funzioni di valutazione quale sia la più efficace.

Capitolo 3

Machine Learning applicato al dominio dei giochi

In questo capitolo vengono passate in rassegna le principali tecniche di ML applicate ai giochi di scacchiera a informazione completa. Sono trattate soprattutto le tecniche che fanno uso di database e che più si prestano ad essere applicate al gioco degli Scacchi. Inoltre non sono riportate tecniche applicate a sottoinsiemi molto ristretti del gioco degli Scacchi, come, ad esempio, gli algoritmi che inducono strategie specifiche per trattare certi tipi di finale (vedi [Mug88] e [TigHer91]). Per ciascuna tecnica vengono anche forniti alcuni esempi, con i relativi risultati sperimentali.

3.1 Apprendimento per memorizzazione

L'apprendimento per memorizzazione è il metodo più semplice di apprendimento. Ciascuna posizione incontrata viene memorizzata assieme al valore che il sistema gli attribuisce sulla base alla propria esperienza.
Un esempio di memorizzazione di posizioni si può trovare in [Sam59], applicata al gioco della Dama (Checkers). Il programma di Samuel memorizzava ogni posizione esaminata durante il gioco con il relativo valore calcolato da un algoritmo di tipo MiniMax. I valori così memorizzati potevano essere usati in seguito, al posto della funzione di valutazione statica, quando il programma trovava una posizione già esaminata. In questo modo si ottiene un aumento della profondità di ricerca.

Applicazioni di questa tecnica agli Scacchi si hanno nel programma Bebe [SST90] e nel programma GnuChess. In entrambi i casi si fa uso di tabelle delle trasposizioni in memoria permanente in cui vengono raccolte delle posizioni analizzate durante il gioco. Le tabelle delle trasposizioni in memoria permanente differiscono sostanzialmente da quelle illustrate nel paragrafo 2.3.2, in quanto quest'ultime non utilizzano le informazioni ricavate da analisi svolte in partite precedenti, ma solo dalle analisi svolte nella partita in corso. Negli esperimenti Bebe è passato dal 55% al 70% di punti fatti, con un determinato avversario, dopo 120 partite, quindi un rafforzamento stimabile in oltre 100 punti Elo. Dato il relativamente basso numero di partite non sono stati previsti meccanismi per la cancellazione delle entry della tabella poco frequentemente utilizzate: è stata sufficiente una tabella di 8000 posizioni. Gnuchess, invece, segue un criterio di rimpiazzamento delle entry della tabella che tiene conto della profondità dell'albero di gioco visitato per stabilire il valore di una posizione: la posizione con profondità maggiore ha la precedenza. In questo modo la tabella contiene posizioni con analisi di profondità media crescente nel tempo. Un'altra strategia per ridurre il numero di posizioni da memorizzare è scegliere solo quelle che hanno subito un forte cambiamento di valutazione tra due iterazioni successive dell'algoritmo di analisi dell'albero di gioco. L'idea di questa tecnica è registrare solo le posizioni per le quali è utile avere un *look-ahead* maggiore durante la ricerca [Sla87].

L'handicap principale dell'apprendimento per memorizzazione è la totale assenza di generalizzazione, cioè la conoscenza acquisita su di una posizione non dice niente sulle altre posizioni. Per questo l'insieme di posizioni già valutate deve essere molto esteso, in rapporto al numero delle posizioni possibili che il giocatore può incontrare, per avere dei sensibili benefici.

Pertanto questo metodo può essere applicato con pieno successo solo su giochi semplici, cioè con un numero trattabile di configurazioni possibili. Per giochi più complessi (per es. il Go può presentare circa $3^{361}/4 \approx 4{,}35 \cdot 10^{171}$ configurazioni diverse, per gli Scacchi se ne stimano circa 10^{43}) tale metodo può essere utile solo in piccoli sottoinsiemi del gioco (per es. l'apertura o il finale negli Scacchi) o comunque solo nei casi in cui si presume che una stessa configurazione possa ricorrere relativamente spesso in partite diverse.

3.2 Reti Neurali

Diversi tentativi sono stati fatti di sostituire la funzione di valutazione di un programma di gioco con una rete neurale in grado di migliorarsi.
La struttura di queste reti è in genere costituita da più nodi di ingresso, anche dell'ordine delle centinaia, rappresentanti lo stato del gioco, e da un solo nodo di uscita, che contiene il valore attribuito alla posizione oppure alla mossa in input. Possono essere presenti anche dei nodi intermedi (detti anche nodi nascosti) tra quelli in ingresso e quelli di uscita.
Le rete neurale così strutturata viene allenata per dare in output sul nodo di uscita approssimazioni sempre migliori della valutazione della posizione che viene rappresentata sui nodi di input. A questo scopo vengono usate posizioni campione di cui già si conosce la valutazione corretta, oppure si possono usare le posizioni giocate dallo stesso giocatore che apprende contro un avversario allenatore. In quest'ultimo caso la valutazione della posizione viene derivata dall'esito finale della partita. Mediante l'uso di appositi algoritmi la rete viene progressivamente modificata in modo che la funzione da essa calcolata si adatti agli esempi esaminati.

3.2.1 Reti neurali nel Backgammon

In [Tes88] è descritta un'applicazione di reti neurali al Backgammon. La rete è costituita da 459 unità di input che codificano la posizione corrente e la mossa da esaminare più alcune euristiche precalcolate, un nodo di uscita contenente il valore relativo attribuito alla mossa in esame, e uno o due livelli di unità nascoste, ciascuna con 12 o 24 unità. La generazione delle mosse legali viene effettuata esternamente alla rete. Ciascuna mossa viene presentata separatamente in input alla rete, e successivamente viene scelta quella con valore relativo più alto.
Il sistema è stato allenato sulla base di 3202 posizioni, a ciascuna delle quali è assegnato un valore dei dadi, e a quasi tutte le mosse legali è associato un punteggio da -100 a +100, attribuito da giocatori esperti. Alle mosse senza punteggio ne viene attribuito uno casuale negativo, presupponendo che non siano mosse buone.

Per l'apprendimento è stato usato un algoritmo di "back-propagation" standard (vedi [Hut94], pp. 118-124).
Durante la sperimentazione si è rilevata determinante la codifica dell'input della rete: una rappresentazione elementare della scacchiera non è sufficiente per comprendere certi princìpi strategici del gioco, mentre uno schema di rappresentazione più complesso sembra essere più indicato.
I risultati finali sono stati sostanzialmente soddisfacenti. Il sistema è stato in grado di battere un programma di livello intermedio in circa il 60% delle partite, mentre contro giocatori umani esperti ha ottenuto il 35%. Questi valori calano quando non vengono usate unità nascoste, cioè nodi intermedi tra i nodi di ingresso e quelli di uscita, e quando in input non vengono inserite euristiche precalcolate.

3.2.2 Reti neurali negli Scacchi

NeuroChess [Thr95] è un programma che impara attraverso l'osservazione di partite giocate contro se stesso e partite giocate da grandi maestri. L'algoritmo di ricerca è quello di GnuChess, ma la funzione di valutazione è sostituita da una rete neurale.
NeuroChess è un'applicazione del metodo dell'"explanation-based neural network" (EBNN).
Il sistema NeuroChess ha due reti neurali, M e V, ciascuna delle quali ha in input 175 caratteristiche della posizione corrente sulla scacchiera. M è un modello del gioco degli Scacchi che predice il valore del vettore di input due semimosse dopo. M viene allenato su partite di grandi maestri per imparare dipendenze temporali tra le caratteristiche della posizione. Ad esempio può apprendere che se un Cavallo può dare un attacco doppio a Donna e Re, dopo due semimosse la Donna verrà catturata.
V implementa la funzione di valutazione ed impara da ogni posizione di ogni partita che le viene presentata, attraverso un metodo di *temporal difference*. Questo metodo fornisce una funzione di valutazione, chiamata *funzione target*, così definita ricorsivamente: per la posizione finale si prende come valutazione il risultato della partita (codificato con 1, 0, -1 rispettivamente la vittoria del Bianco, la patta, la vittoria del Nero), per le altre si prende la valutazione della posizione due semimosse avanti moltiplicata per un coefficiente compreso nell'intervallo reale [0,1]. Inoltre V tiene conto anche della derivata della

funzione target al variare delle caratteristiche della posizione. Questa derivata viene approssimata con la derivata di M rispetto al vettore di input. L'uso di questa derivata permette un apprendimento più veloce e accurato, in quanto permette di individuare quali sono le caratteristiche più rilevanti in una posizione e quali lo sono meno ai fini della valutazione. Questo è confermato dai risultati: dopo un'allenamento di M su 120.000 partite e di V su 2.400, NeuroChess batte GnuChess circa il 13% delle volte, mentre senza l'uso di M questa percentuale scende al 10%.
Durante la sperimentazione è stato osservato che apprendere esclusivamente da partite contro se stesso non porta a risultati soddisfacenti. D'altra parte, usare prevalentemente partite di grandi maestri crea delle conoscenze erronee, specie in fase di apertura e finale, che portano il sistema a giocare anche mosse estremamente deboli. Forse questo è dovuto al fatto che il sistema apprende dei princìpi strategicamente validi, ma non sempre li applica nelle situazioni più idonee (per es. centralizzare il re va bene in genere in finale di partita, quando non ci sono rischi di matto, ma non prima).
Il limite di NeuroChess sembra essere proprio questa discontinuità di gioco forse dovuta a un tempo di apprendimento non sufficiente oppure a una scelta delle caratteristiche in input non adeguata.

3.2.3 *Reti neurali per l'apprendimento di qualsiasi gioco*

Il programma SAL (Search And Learning) [Ghe93] generalizza l'apprendimento con reti neurali a qualsiasi gioco ad informazione completa. In SAL il meccanismo di apprendimento non cambia da gioco a gioco, l'unica cosa di cui SAL ha bisogno sono le regole del gioco, che gli vengono fornite tramite il generatore di mosse.
La struttura della rete neurale impiegata in SAL prevede un insieme di nodi di ingresso con la rappresentazione della posizione corrente, un livello di nodi intermedi, connessi con tutti i nodi di ingresso, costituito da un nodo ogni dieci nodi di ingresso, e un nodo di uscita connesso con i nodi intermedi.
Dato che la rete può apprendere funzioni di valutazione per giochi diversi, nei nodi di input non sono inserite euristiche precalcolate utili per un gioco in particolare, ma solo euristiche che hanno significato indipendentemente dal particolare gioco cui la rete è applicata. Ad esempio tra le euristiche nei nodi di ingresso sono presenti, oltre alla

semplice rappresentazione della posizione sulla scacchiera, il numero e il tipo di pezzi catturati, la possibilità di perdere la partita alla prossima mossa, i pezzi che possono essere catturati, ecc.
La fonte di apprendimento sono tutte le posizioni di partite giocate da SAL stesso contro un programma allenatore, ovviamente diverso per ogni gioco al quale viene applicato. Ogni posizione viene valutata semplicemente con il risultato finale della partita. La modifica della rete avviene tramite un algoritmo di "back-propagation" (vedi [Hut94], pp. 118-124).
SAL ha dato buoni risultati con un gioco semplice come il Tic-Tac-Toe arrivando a giocare perfettamente dopo 20.000 partite di allenamento. Negli Scacchi invece non è riuscito a vincere nessuna partita su oltre 4.000 partite di allenamento giocate contro GnuChess. Nonostante questo l'effetto dell'apprendimento è osservabile dal progressivo aumentare del numero di mosse necessarie in ogni partita a GnuChess per sconfiggere il suo avversario.

3.3 Apprendimento Bayesiano

L'apprendimento Bayesiano è basato sul teorema di calcolo delle probabilità di Bayes, che si può sintetizzare nella formula:

$$p(H|D) = \frac{p(H)p(D|H)}{p(D)} \quad (1)$$

dove con D si indicano i dati osservati e con H una ipotesi. La formula dice che la probabilità che l'ipotesi sia vera, conosciuti i dati ($p(H|D)$, detta anche *probabilità a posteriori dell'ipotesi*), è uguale al prodotto della probabilità inerente dell'ipotesi ($p(H)$, detta *probabilità a priori dell'ipotesi*) per la probabilità di osservare i dati conosciuta l'ipotesi ($p(D|H)$ detta *verosimiglianza* o *likelihood* dei dati) diviso la probabilità inerente dei dati $p(D)$.
Lo scopo degli algoritmi basati su apprendimento Bayesiano è, in genere, classificare gli elementi di un database tramite l'osservazione

dei loro attributi. In questo caso l'evento H della formula (1) si può quindi interpretare come il numero e la descrizione delle classi secondo le quali gli elementi del database sono divisi.
L'appartenenza di un elemento x_i del database ad una classe C_j non viene stabilita dall'algoritmo in maniera certa, ma ne viene data la probabilità, usando sempre il teorema di Bayes:

$$p(x_i \in C_j | x_i, \vec{\theta}, \vec{\pi}, J) = \frac{\pi_j p(x_i | x_i \in C_j, \vec{\theta}_j)}{p(x_i | \vec{\theta}, \vec{\pi}, J)} \quad (2)$$

dove J è il numero complessivo delle classi, $\vec{\theta}$ il vettore dei descrittori delle distribuzioni degli attributi per ciascuna classe, e $\vec{\pi}$ il vettore delle probabilità delle classi, per cui si ha $\pi_j = p(x_i \in C_j)$. Da ciò deriva che un elemento può far parte di classi diverse con probabilità diverse.

3.3.1 Apprendimento Bayesiano nell'ambito dei giochi

Nel campo dei giochi l'apprendimento Bayesiano è stato usato nel programma Bill, un giocatore artificiale di Othello sviluppato da K.F. Lee e S. Mahajan [LeeMah90]. La sua struttura è quella ormai universalmente affermatasi per i giochi a informazione completa: algoritmo AlphaBeta, iterative deepening, tabelle hash e killer, ecc. Della funzione di valutazione, che è la parte del programma che più ci interessa, sono state create due versioni: Bill 2.0 calcola 4 diverse euristiche (mobilità, stabilità degli angoli, mobilità potenziale, case occupate) e le combina linearmente in un unico valore; Bill 3.0 calcola le stesse euristiche, che però vengono date in input ad una funzione risultante dall'apprendimento Bayesiano che restituisce la probabilità che la posizione in esame sia vincente.
L'algoritmo di apprendimento si svolge attraverso le seguenti fasi:

1. Generare di un grande database di posizioni di esempio.
2. Etichettare queste posizioni come vincenti o perdenti attraverso una ricerca esaustiva (questo è possibile in Othello perché il numero di semimosse di una partita è al massimo 60). Le patte vengono scartate.
3. Determinare una funzione discriminante a partire dalle posizioni di esempio. Con riferimento alla (2) ciò equivale a stabilire $\vec{\theta}$. Gli autori hanno scelto di approssimare la distribuzione dei valori determinati dalle euristiche di valutazione con leggi normali multiva-

riate. Perciò ciascun elemento $\dot{\theta}_i$ del vettore $\dot{\theta}$ è costituito da una coppia $(\dot{\mu}_i, A_i)$ formata dal vettore delle medie e dalla matrice di covarianza derivate statisticamente dai dati ricavati dalle posizioni di esempio appartenenti alla classe i-esima.

La fase 3 viene ripetuta per i diversi stadi del gioco (in Othello definiti dal numero di mosse effettuate).

Il miglioramento ottenuto dopo l'apprendimento è stato notevole: Bill 3.0 ha totalizzato, contro Bill 2.0, 139 vittorie, 6 patte e 55 sconfitte, cioè quasi il 70% di vittorie. Un incremento di prestazioni paragonabile a quello ottenibile con l'aumento della profondità di ricerca di 2 semi-mosse. Gli autori attribuiscono questo risultato alla capacità della nuova funzione di valutazione di percepire relazioni non lineari tra le varie euristiche, grazie alla matrice di covarianza.

3.4 Algoritmi Genetici

Gli algoritmi genetici sono procedimenti di ottimizzazione che traggono ispirazione dalla teoria della selezione naturale di Darwin. La loro definizione può essere trovata in [Bag67] e [Hol19], mentre una rielaborazione per il ML si può trovare in [Gol89]. L'idea di base è quella di applicare ad una popolazione di individui (nel nostro caso programmi) le leggi che regolano l'evoluzione delle specie.

Gli elementi fondamentali di un algoritmo genetico sono:

- Un sistema di codifica degli individui: ad ogni individuo è associato un codice genetico (una stringa di bit) che ne determina il comportamento, ovvero la soluzione che il singolo individuo darà al problema in esame.
- Una funzione di adattamento (*fitness function*), che misura il grado di rispondenza di ciascun individuo al problema da affrontare.
- Un meccanismo di riproduzione, che fa sì che a individui con grado di adattamento più elevato corrisponda una maggiore probabilità di trasmettere il proprio codice genetico alla generazione successiva.

Una volta creata la popolazione iniziale, l'algoritmo ripete una fase di calcolo della *fitness function* per tutti gli individui, ed una di creazione della popolazione della generazione successiva, accoppiando tra loro gli individui più adatti (fig. 3.1).

Creazione della popolazione iniziale

Calcolo della Fitness Function per tutti gli individui

Creazione della generazione successiva

Figura 3.1. Schema di base di un algoritmo genetico

3.4.1 Algoritmi genetici negli Scacchi

In [Tun91] si ha un esempio di algoritmo genetico applicato alla funzione di valutazione di un giocatore artificiale di Scacchi. Lo scopo dell'apprendimento è ottimizzare alcune variabili della funzione di valutazione, tra le quali i pesi delle diverse euristiche usate (materiale, mobilità, controllo del centro, sviluppo, ecc.). Di ogni variabile viene stabilito a priori un campo di variabilità, in modo da assegnare ad ognuna di esse il minor numero possibile di bit nella stringa che rappresenta il codice genetico (fig. 3.2).

controllo del centro   valore di un alfiere   eccetera ...

0 1 0 0 1 1 0 1 0 1 1 1 0 1 0 1 1 0 0 1 0 1 0 1 .. 0 1 0 0

145 bits

Figura 3.2. Esempio di codice genetico di un giocatore artificiale.

Come *fitness function* è stata utilizzata una procedura che esamina 500 posizioni scelte casualmente in un database di partite di grandi maestri; il numero di posizioni in cui il giocatore artificiale sceglie la stessa mossa giocata in partita è il risultato della funzione.
L'esperimento è stato condotto con una popolazione di 50 individui. Dopo 100 generazioni l'algoritmo è sembrato stabilizzarsi, il valore medio della *fitness function* è passato da 82 mosse su 500 scelte corret-

tamente, nella prima generazione (generata casualmente), a 107 su 500, dimostrando un significativo miglioramento del gioco del programma.
L'esperimento descritto in [San93] è analogo al precedente: lo scopo è ottimizzare i pesi da attribuire a otto euristiche di valutazione impiegate, questa volta, in un programma di gioco parallelo a conoscenza distribuita. La *fitness function* anche qui è un test su 500 posizioni, mentre le popolazioni sono costituite da gruppi di 30 individui. Il miglioramento riscontrato è stato il passaggio da 88 a 104 mosse scelte correttamente su 500.

3.5 Regressione lineare

Per calibrare i pesi delle euristiche di valutazione parziali sono stati impiegati anche metodi per la soluzione di problemi di regressione lineare ([Bal92], pp. 220-227). Questi metodi sono stati applicati soprattutto al gioco degli Scacchi. L'idea è sostanzialmente la stessa degli algoritmi genetici: calibrare i pesi delle euristiche in modo che il maggior numero di mosse scelte dal programma, per un insieme di posizioni di test, siano identiche a quelle giocate da maestri.
In [Nit82] e [Ana90] sono descritte due diverse applicazioni di tecniche di regressione lineare. In [Nit82] si fa l'ipotesi che ogni mossa venga scelta con una ricerca di una sola semimossa di profondità, per cui il problema di trovare una combinazione di pesi con la quale il maggior numero di mosse scelte dal programma corrisponda a quelle dei maestri è facilmente riducibile ad un problema di regressione lineare. L'autore riporta solo un esempio di applicazione di questa tecnica, con successo, al finale Donna e Re contro Re, dove le euristiche di valutazione sono molto particolari, diverse da quelle usate nella maggior parte delle situazioni. L'impressione è che la ricerca di una sola semimossa di profondità dia un'approssimazione troppo rozza della valutazione della posizione, specie nel caso di posizioni turbolente.
In [Ana90] (pp. 20-21) la ricerca viene estesa con una ricerca quiescente oltre la prima semimossa. Ciò implica che la mossa scelta dal programma non dipenda più linearmente dai pesi delle euristiche di valutazione, ma in maniera più complessa. Per ridurre il problema ad un problema di regressione lineare, si assume che il valore di ciascuna mossa al top level dell'albero di gioco parziale sia determinato linearmente dai valori delle euristiche del relativo nodo foglia che ne

determina il valore minimax, detti nodi dominanti. Una volta fatta questa semplificazione si può risolvere il problema di regressione lineare che ne deriva. Cambiando però i pesi delle euristiche di valutazione, dopo la soluzione del problema di regressione lineare, i nodi dominanti possono non essere più gli stessi, perciò l'algoritmo viene iterato più volte. Negli esperimenti effettuati si è osservata una oscillazione dei valori dei parametri da ottimizzare, abbastanza comprensibile proprio perché si tenta di risolvere con un metodo lineare un problema di per sé non lineare.
Un'altra tecnica per bilanciare i pesi della funzione di valutazione è descritta in [Meu89]. Anche in questo caso la mossa scelta dal programma per ogni posizione campione viene determinata con una ricerca di una sola semimossa di profondità, senza ricerca quiescente. Per ogni posizione campione viene determinato, tramite la risoluzione di un sistema di disuguaglianze, l'insieme dei vettori dei pesi che portano alla scelta della mossa corretta. Di questi insiemi viene poi calcolata l'intersezione, che è l'insieme delle combinazioni di pesi ottime per tutte le posizioni campione. È verosimile che questa intersezione sia vuota se le posizioni sono molto diverse tra loro, perciò vengono individuati diversi sottoinsiemi di posizioni a cui applicare combinazioni di pesi diversi. L'autore non riporta esempi concreti di applicazione di questa tecnica, ma come in [Nit82] la ricerca non quiescente che viene utilizza sembra essere il maggior difetto.

3.6 Riconoscimento di pattern

Un metodo per analizzare database di partite o di posizioni è di individuare in ciascuna posizione del database un insieme di pattern per ricavare informazioni utili su posizioni simili, che hanno cioè un certo numero di pattern in comune.
In [GeoSch88] è descritto un sistema denominato MACH (Master Advisor for CHess) che realizza questa idea. I pattern (o *chunk*) riconosciuti dal sistema sono forniti da maestri di Scacchi e descrivono la posizione di alcuni pezzi sulla scacchiera. A MACH è fornito una database di partite che viene analizzato in base ai pattern definiti. Data una posizione, MACH è in grado di individuare quali posizioni simili sono presenti nel database e quali mosse sono state giocate in queste posizioni. Per stabilire quando due posizioni sono simili si confrontano i loro pattern: se la posizione che vogliamo cercare soddisfa L pattern,

vengono considerate simili le posizioni che soddisfano un sottinsieme di questi composto da almeno L-1 elementi.
MACH è stato usato per dare suggerimenti al programma di gioco Phoenix. Alle mosse suggerite da MACH viene assegnato un bonus rispetto alla valutazione effettuata da Phoenix. Secondo gli autori il compito principale di MACH è di aiutare il programma di gioco nella fase immediatamente successiva all'apertura, fase in cui spesso i giocatori artificiali non rispettano la linea strategica imposta loro dal libro di apertura. Gli esperimenti hanno confermato che frequentemente MACH suggerisce una mossa migliore di quella calcolata da Phoenix.

3.7 Approcci ibridi

3.7.1 Il progetto Morph

Struttura del giocatore artificiale
Il progetto Morph [Lev91] si differenzia sostanzialmente da quelli visti finora. Prima di tutto la struttura di base di base di Morph non è quella classica con visita dell'albero di gioco e funzione di valutazione statica. Morph infatti esegue una ricerca di una sola semimossa; la valutazione delle posizioni generate viene eseguita tramite il riconoscimento di pattern, rilevati nelle posizioni stesse, e la combinazione di pesi associati ai pattern. Morph tratta due tipi di pattern: differenza di materiale (per es. "un Pedone di vantaggio", "un Cavallo di svantaggio" sono pattern di questo tipo) e relazione tra i pezzi sulla scacchiera. I pattern di questo secondo tipo sono grafi diretti in cui i nodi sono i pezzi, oppure alcune case particolari (per es. le case vicine al Re), e gli archi rappresentano le relazioni di attacco tra i pezzi, distinguendo tra attacchi diretti e attacchi indiretti.
I pesi associati ai pattern sono valori compresi nell'intervallo [0,1]. Questi pesi possono essere interpretati come il valore minimax delle posizioni in cui il pattern compare. Dato che in una posizione compaiono verosimilmente più pattern, il valore assegnato alla posizione in esame viene calcolato tramite una apposita funzione che ha in input i pesi associati a tutti i pattern riscontrati nella posizione stessa.

Sistema di apprendimento
L'apprendimento in Morph avviene attraverso l'osservazione di proprie partite giocate contro un avversario. In particolare è stato usato GnuChess che gioca facendo ricerche ad una profondità di una sola semimossa. Dopo ogni partita vengono creati dei pattern e altri cancellati, e vengono modificati, sulla base del risultato finale della partita, i pesi associati a dei pattern per rendere la propria capacità di valutazione più accurata. Periodicamente alcuni pattern vengono cancellati perché considerati poco determinanti ai fini del risultato oppure perché si presentano con frequenza trascurabile.
Morph è un giocatore artificiale in cui viene enfatizzato l'uso di diversi metodi di Machine Learning combinati tra loro.
Per l'aggiustamento dei pesi dei pattern alla fine di ogni partita viene utilizzato il metodo del temporal difference learning (TD): con questo, sulla base del risultato finale della partita, viene costruita una funzione target che stima la valutazione di tutte le posizioni occorse durante la partita, quindi i pesi associati ai pattern presenti in queste posizioni vengono modificati in modo da avvicinare la valutazione di Morph alla funzione target. Congiuntamente a questo procedimento di TD viene impiegata la tecnica del *simulated annealing*. Questa tecnica tiene conto, ogni volta che un peso viene variato, di quanti aggiustamenti sono stati fatti, più aggiornamenti sono stati fatti e minore è l'entità variazione del peso che viene seguita. Così i pesi, dopo i primi aggiustamenti, tendono a variare in maniera continua e a convergere più rapidamente.
In Morph vengono anche usati algoritmi genetici per decidere quali memorizzare tra gli innumerevoli pattern. La popolazione (dal punto di vista dell'algoritmo genetico) è costituita dai pattern, mentre la fitness function definita su di essi privilegia quelli che vengono incontrati spesso, con poca varianza del peso a loro associato, e con valori estremi (vicini a 0 o a 1). Sono presenti anche dei parametri per tenere basso il numero di pattern. L'algoritmo genetico inserito in Morph è stato opportunamente adattato per trattare i pattern, che non hanno lunghezza fissa e non sono ordinati, al contrario delle stringhe di bit su cui operano normalmente gli algoritmi genetici.
Infine, viene anche utilizzata una forma di explanation based generalization. Questo meccanismo stabilisce relazioni di precedenza tra i pattern, trovando pattern che sono precondizioni di altri, osservando le partite giocate da Morph contro GnuChess. L'idea è cercare sequenze di pattern che portano a posizioni con valori estremi, e rendere estremi

anche i pesi dei pattern che ne sono le precondizioni, così il programma è invogliato ad entrare in certe sequenze di mosse e a non entrare in altre.

Risultati
Nonostante lo spiegamento di tutte queste tecniche di Machine Learning, Morph non è riuscito ad andare oltre 1 vittoria e 20 patte, su un totale di ben 3000 partite, contro un avversario modesto come GnuChess limitato ad una sola semimossa di ricerca. Qualcosa comunque il sistema è riuscito ad apprendere. Ad esempio, dopo una trentina di partite i pesi attribuiti ai pattern riguardanti le differenze di materiale assumono valori in linea con la teoria scacchistica.
Il metodo del riconoscimento di pattern non sembra poter competere contro le usuali funzioni di valutazione statica delle posizioni.
Tra gli sviluppi che gli autori intendono apportare in Morph è da notare l'utilizzo di database di partite (in alternativa alle partite giocate da Morph stesso), in particolare di database riguardanti la carriera di un singolo giocatore, in modo da ripercorrere le tappe dell'apprendimento del giocatore umano.

3.8 Considerazioni generali

Confrontando tra loro i diversi approcci al dominio dei giochi ad informazione completa attraverso tecniche di Machine Learning, e i relativi risultati, possiamo avere un'idea dei vantaggi e degli svantaggi di ciascuna tecnica.
Abbiamo visto che le tecniche più semplici, basate sulla memorizzazione, sono adatte per giochi semplici. Al contrario, per i giochi più complessi, come nel caso degli Scacchi, di cui ci occuperemo, l'apprendimento per memorizzazione non può costituire la via principale per inserire conoscenza nei giocatori artificiali, a causa della assoluta mancanza di generalizzazione della conoscenza a situazioni diverse da quelle apprese durante l'allenamento. Da ciò deriva che le quantità di informazioni e di tempo necessari, in fase di apprendimento, per raggiungere un buon livello di gioco, con il solo uso di questa tecnica, superano ogni limite ragionevole. Tuttavia, l'apprendimento per memorizzazione può intervenire con successo in sottoinsiemi del gioco, come la fase di apertura, lasciando ad altri metodi di apprendi-

mento, con maggiore capacità di generalizzazione, il compito di sintetizzare conoscenze utili per le situazioni più generali.
Tra le tecniche di apprendimento con maggiore capacità di generalizzazione è stato molto sperimentato l'uso di reti neurali. Tuttavia, soprattutto per quanto riguarda gli Scacchi, i risultati ottenuti dai programmi di gioco con la funzione di valutazione implementata con una rete neurale sono molto lontani da quelli ottenuti con la classica funzione di valutazione costruita senza tecniche di ML. Osservando i risultati degli esperimenti descritti nel paragrafo 3.2, sembra che l'apprendimento di una funzione di valutazione di un gioco complesso come gli Scacchi sia aldilà delle capacità delle reti neurali, soprattutto se la rappresentazione della posizione fornita al sistema non contiene euristiche significative già precalcolate, ma solo la disposizione dei pezzi sulla scacchiera.
Gli algoritmi genetici si dimostrano validi metodi di ottimizzazione per le funzioni di valutazione. Inoltre hanno il vantaggio di essere facilmente implementabili e di poter ottimizzare qualsiasi insieme di parametri. L'handicap principale di questa tecnica è, però, l'enorme quantità di tempo macchina necessaria. In particolare, nell'ambito dei giocatori artificiali, la fase più dispendiosa è il calcolo della *fitness function*. L'ideale sarebbe poter confrontare in un torneo tutti gli appartenenti ad una popolazione, ma ciò è improponibile per l'elevato costo in termini di tempo (per es. con popolazioni di 50 individui effettuare un torneo all'italiana costa oltre 100 ore, considerando solo 5 minuti per partita). Usare i test su posizioni campione, come negli esempi precedenti, non risolve il problema: nel caso dell'esperimento di Tunstall-Pedoe sono sempre necessarie 4 ore per generazione, una quantità di tempo considerevole, che va poi moltiplicata per il numero di generazioni necessarie prima che il procedimento si stabilizzi. Questo è un ostacolo ancora maggiore quando si vogliano ottimizzare molti parametri, oppure ottimizzare diverse funzioni di valutazione.
Al contrario degli algoritmi genetici, l'apprendimento Bayesiano ha un costo computazionale assai più limitato in quanto è necessario solo valutare una volta ogni posizione campione. Tuttavia l'efficacia di questo metodo di apprendimento nel dominio dei giochi non è stata molto sperimentata. Comunque nell'unico gioco in cui questo metodo è stato applicato, l'Othello, i risultati sono stati molto positivi.
Nei prossimi capitoli saranno presentati alcuni esperimenti di ML che usano come fonte di apprendimento database di partite, prevalentemente tra giocatori umani di alto livello, non commentate. Da questo

tipo di database ci si aspetta di poter estrarre conoscenza utile per un giocatore artificiale che può apprendere dalle mosse giocate da forti giocatori. In particolare queste conoscenze riguarderanno soprattutto le fasi di apertura e mediogioco, e meno la fase finale della partita, che molto spesso, in partite tra forti giocatori, non viene giocato, essendo già chiaro dal mediogioco quale sarà l'esito finale della partita. Saranno quindi utilizzate tecniche di apprendimento per memorizzazione, che è particolarmente adatto per le fasi iniziali del gioco. Per il mediogioco, invece, saranno sperimentati l'apprendimento Bayesiano e gli algoritmi genetici, attraverso queste tecniche si tenterà di migliorare la funzione di valutazione statica di GnuChess 4.0. Queste sono le tecniche che per il momento sembrano poter dare i migliori risultati, anche se l'apprendimento Bayesiano è stato sperimentato finora solo sul gioco dell'Othello, e non con database di partite di giocatori umani come fonte di dati per l'apprendimento.

Capitolo 4

Creazione automatica del libro di aperture

4.1 Introduzione

L'apertura è la fase della partita che inizia dalla prima mossa e approssimativamente finisce quando entrambi i contendenti hanno completato lo sviluppo dei propri pezzi. Dato che le posizioni di questa fase possono ricorrere in molte partite, durante i secoli si sono studiate a fondo le prime mosse della partita e si è sviluppata una vasta letteratura in continua espansione. Avvalendosi di questi studi, giocatori ben preparati possono giocare l'apertura in modo apparentemente automatico. In questo modo si evitano brutte sorprese nelle prime mosse e viene risparmiato del tempo di riflessione per le mosse successive, quando la partita inevitabilmente uscirà dalle varianti conosciute. Se questo comportamento è valido per un giocatore umano, ancora di più lo è per un giocatore artificiale, le cui capacità mnemoniche sono assai maggiori.
Per un giocatore artificiale un libro di aperture è un archivio in cui può trovare direttamente la mossa, o un insieme di mosse tra cui scegliere, da giocare in determinate situazioni. Attualmente esistono in commercio programmi che sfruttano pesantemente questa capacità di archiviazione con libri di aperture che contengono centinaia di migliaia di mosse e con varianti che possono superare le 30 mosse di profondità. Tuttavia un libro di aperture non si misura solamente con la sua dimensione, occorre anche che le varianti in esso contenute siano solide e non in contrasto con i princìpi strategici del giocatore artificiale. Può capitare altrimenti che il programma giochi inconsapevolmente le mosse di apertura per raggiungere determinati obiettivi (per

esempio può sacrificare un Pedone per ottenere l'iniziativa) e al termine dell'apertura, quando viene attivata la funzione di valutazione, giochi per ottenere gli obiettivi opposti (per esempio riconquistare il Pedone sacrificato a costo di perdere l'iniziativa). È chiaro che una condotta di gioco così schizofrenica non può portare, nella maggior parte dei casi, a buoni risultati.
Dal punto di vista realizzativo possiamo distinguere due diversi tipi di libri di aperture: quelli che gestiscono le trasposizioni e quelli che non le gestiscono.

4.1.1 Libri senza gestione delle trasposizioni

Nei libri di aperture che non gestiscono le trasposizioni la struttura dati utilizzata è generalmente un albero. Ai nodi corrispondono le posizioni sulla scacchiera e agli archi corrispondono le mosse. La posizione iniziale si trova nella radice dell'albero. Il giocatore artificiale segue lo svolgimento della partita lungo l'albero delle aperture: scegliendo uno degli archi uscenti dalla posizione corrente come mossa da giocare quando è il suo turno, e selezionando la nuova posizione corrente quando la mossa è effettuata dall'avversario. L'utilità del libro termina appena si raggiunge un nodo foglia dell'albero, oppure quando l'avversario gioca una mossa non prevista nel libro per la posizione corrente.
Questo tipo di libro è molto semplice da implementare [Tak91] e permette soluzioni molto efficienti dal punto di vista del consumo di memoria (1 byte per mossa, vedi [Cia92] pag. 94).
Il problema di questo tipo di libro di aperture è che due posizioni identiche, ma generate da diversi ordini di mosse, possono non venire trattate uniformemente: una può essere presente nel libro di aperture e l'altra no, oppure attribuite a nodi diversi dell'albero, mentre sarebbe più corretto trattare entrambe nella stessa maniera. Dato che nella pratica scacchistica le inversioni di mosse sono piuttosto frequenti, questo problema non può essere trascurato senza una perdita di efficacia del libro di aperture.
Un'idea delle conseguenze cui può portare la mancanza della gestione delle trasposizioni la dà l'errore commesso dal prototipo di Deep Blue contro Fritz nel campionato mondiale per giocatori artificiali svoltosi a Hong Kong nel maggio del 1995 [Bea95]. Nella posizione del diagramma della figura 4.1, Deep Blue (che gioca col Bianco) si aspettava 12. ... Ag7

e poi successiva 13. ... f4, ma Fritz giocò subito 12. ... f4, una mossa ben nota alla teoria. Invece Deep Blue si trovò fuori dal suo libro di aperture e non giocò in modo da entrare nella possibile trasposizione ottenibile con 13. c3 (o 13. g3) Ag7. Fritz vinse poi dopo altre 27 mosse.

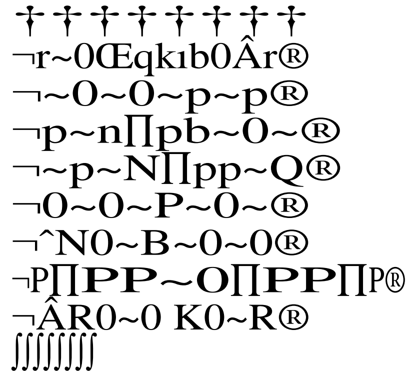


Figura 4.1. Deep Blue - Fritz. Posizione dopo 12. Dh5.

4.1.2 Libri con gestione delle trasposizioni

I libri di aperture che gestiscono le trasposizioni superano l'evidente limite di quelli del tipo visto in precedenza. Per superare questo problema occorre avere un archivio non più strutturato ad albero. Occorre una struttura dati che permetta in maniera efficiente di stabilire se una posizione appartiene o no alle posizioni di libreria. Per esempio, la soluzione adottata da GnuChess è di utilizzare la propria tabella delle trasposizioni: all'inizio della partita in corrispondenza delle entry relative a ciascuna posizione di libreria viene inserita la mossa suggerita dal libro delle aperture. Questa informazione viene utilizzata nel caso la posizione del libro si presenti durante la partita mentre è il turno di GnuChess (vedi appendice A.4).

4.2 Creazione di un libro di aperture

I libri di apertura dei giocatori artificiali normalmente vengono costruiti a partire dalla letteratura scacchistica destinata ai giocatori umani e dall'esperienza di qualche esperto che cerca di scegliere le varianti da inserire nel libro in base alle caratteristiche di gioco del programma e, quando si conoscono, di eventuali avversari. Questo

modo di procedere evidentemente richiede molto tempo e esperienza per ottenere buoni risultati. Sarebbe più comodo riuscire a costruire in maniera automatica il libro di aperture tramite l'osservazione di partite giocate ad alto livello. Ciò permetterebbe al programma di acquisire il libro si aperture senza bisogno dell'intervento diretto di un giocatore esperto ed inoltre renderebbe più agevole il periodico aggiornamento del libro tramite l'osservazione di nuove partite. La teoria delle aperture infatti è in continua evoluzione e quindi la capacità del programma di poter acquisire rapidamente nuove varianti non è da disprezzare.
Nel resto del capitolo si vedrà come un meccanismo di creazione automatica del libro di aperture sia stato implementato per il programma di gioco GnuChess 4.0, con i dati del database di partite Deja Vu.

4.3 Codifica delle informazioni e scelte di progetto

Il libro di aperture viene realizzato mediante una coppia di tabelle hash (che chiameremo tabelle di apertura): una per le posizioni in cui la mossa è al Bianco, e l'altra per le posizioni in cui la mossa è al Nero. Le tabelle hanno uguale dimensione. Ciascuna posizione presente in qualsiasi partita del database viene inserita nella tabella appropriata; in realtà, per motivi di risparmio di memoria, sono state inserite solo le prime 35 mosse (cioè 70 semimosse) di ciascuna partita. Questo non pregiudica la validità del libro in quanto è rarissimo che una posizione oltre la 35ª mossa di una partita si ripeta esattamente in un'altra, e quindi le posizioni oltre la 35ª mossa, anche se inserite nella tabella hash, avrebbero una probabilità praticamente nulla di essere usate in fase di gioco.
La funzione hash che restituisce l'indirizzo nella tabella per ogni posizione è quella usata da GnuChess 4.0 per accedere alla propria tabella delle trasposizioni (vedi appendice A.3 e [Zob70]), per cui non si ha nessun overhead in fase di gioco per calcolare l'indirizzo di una posizione nel libro di aperture, in quanto questo calcolo viene comunque svolto per accedere alla tabella delle trasposizioni.
Ciascuna entry della tabella di apertura è del tipo:

```
struct tableEntry
{
     unsigned long     hashbd; // codice di controllo
```

```
        unsigned short    whiteV; // vittorie del bianco
        unsigned short    draw;   // patte
        unsigned short    blackV; // vittorie del nero
    }
```

`Hashbd` è un codice di controllo, generato con lo stesso meccanismo della funzione hash, ma con costanti casuali diverse, che serve a distinguere varie posizioni che potrebbero andare in conflitto, avendo lo stesso indirizzo hash. Si noterà che nella entry non compare alcuna rappresentazione della situazione sulla scacchiera. Inserire una rappresentazione completa della posizione, anche utilizzando metodi sofisticati di compressione [Bal94], avrebbe comportato uno spreco maggiore di memoria (circa 6 byte in più per ogni entry) e un costo supplementare, in termini di tempo di elaborazione, per la decodifica e il confronto tra la posizione nella tabella e quella cercata durante la fase di gioco.

4.3.1 Probabilità di errore nell'uso delle tabelle di apertura

Usando una tabella hash per l'implementazione del libro delle aperture si deve tener conto di due tipi di errori che si possono verificare sia in fase di creazione che in fase di lettura della tabella durante il gioco:

- Il primo tipo di errore (chiamato *tipo 1* in [Zob70]) è il più importante. Questo errore si verifica quando due posizioni diverse sulla scacchiera generano il medesimo valore hash. Questa eventualità non può essere evitata a priori, in quanto il numero dei possibili valori che può restituire la funzione hash è assai minore del numero delle possibili configurazioni della scacchiera. Le conseguenze cui può portare questo errore sono evidenti: una posizione (magari pessima per chi ha la mossa) può erroneamente venire considerata appartenente al libro di apertura e quindi forzare il programma a giocare la mossa che porta alla posizione stessa. Per ridurre la probabilità che questo tipo di errore si verifichi occorre incrementare il numero dei possibili valori restituiti dalla funzione hash.

- Il secondo tipo di errore (*tipo 2* in [Zob70]) si verifica quando a due differenti posizioni viene assegnata la medesima entry nella tabella hash (collisione). La frequenza di questo errore è proporzionale alla

percentuale di entry occupate nella tabella; perciò, a parità di dati da immettervi, la probabilità che questo errore si verifichi diminuisce al crescere della dimensione della tabella hash. Per la tabella di apertura è stata usata la soluzione di far slittare una delle due posizioni in conflitto alla entry successiva. Se si ha ancora un conflitto si prova con la successiva ancora, ecc., fino ad un massimo di 20 tentativi. Se per 20 volte non si è trovata una entry libera l'inserimento della posizione nella tabella abortisce.

Considerando questi due tipi di errori si può maggiorare la probabilità di leggere erroneamente i dati di una posizione diversa da quella che si sta cercando nella tabella di apertura con:

$$\frac{R}{N \cdot H} = \frac{20}{2^{19} \cdot 2^{32}} < 2^{-46} \cong 1.4211 \cdot 10^{-14}$$

dove con R si indica il numero massimo di tentativi nella ricerca della posizione in caso di collisioni, con N il numero di entry nella tabella hash, con H il numero dei possibili valori assegnabili alla variabile *Hashbd*. Nella formula sono stati inseriti i valori usati per i successivi esperimenti (R=20, $N=2^{19}$, $H=2^{32}$). La maggiorazione così ottenuta è talmente piccola da poter considerare la lettura della tabella hash praticamente immune da errori.

4.4 Uso della tabella di apertura

Completata la fase di costruzione della tabella di apertura, questa può essere impiegata da GnuChess, o da un altro giocatore artificiale appositamente predisposto, come libro delle aperture. Come abbiamo visto nel paragrafo precedente, la tabella di apertura non contiene informazioni esplicite sulle mosse da giocare, ma solo delle statistiche sui risultati finali ottenuti nelle partite in cui si sono presentate alcune particolari posizioni. GnuChess quindi dovrà in qualche modo usare queste informazioni per determinare le mosse da giocare. L'idea di base è quella di far giocare a GnuChess la mossa che porta alla posizione successiva più "promettente". Per stabilire tra le varie mosse alternative quale sia la più promettente si possono definire svariati metodi euristici che attribuiscono a ciascuna posizione uno score in

base ai dati statistici ricavati dalla tabella di apertura. Alcuni potrebbero essere, data una posizione *p*:

1. Attribuire uno score costante *k*>0 a *p* se è presente nella tabella di apertura, 0 altrimenti, la scelta tra le varie opzioni con lo stesso punteggio viene effettuata con un metodo casuale, in formule:

   $$score(p) = \begin{cases} k & \text{se } p \in \mathrm{H} \\ 0 & \text{altrimenti} \end{cases}$$

   dove H indica l'insieme delle posizioni presenti nella tabella di apertura;

2. Considerare le somme dei punti ottenuti da ciascun colore, in formule:

   $$score(p) = \begin{cases} W(p) + \frac{1}{2}D(p) & \text{se } p \in \mathrm{H} \\ 0 & \text{altrimenti} \end{cases}$$

   dove *W(p)* e *D(p)* restituiscono rispettivamente il numero di vittorie e di patte ottenute dal colore che deve muovere, nelle partite del database dove occorre la posizione *p*;

3. Considerare le percentuali dei punti ottenute da ciascun colore, in formule:

   $$score(p) = \begin{cases} \dfrac{W(p) + \frac{1}{2}D(p)}{W(p) + D(p) + L(p)} & \text{se } p \in \mathrm{H} \\ 0 & \text{altrimenti} \end{cases}$$

   dove *L(p)* restituisce il numero di sconfitte subite dal colore che deve muovere, nelle partite del database dove occorre la posizione *p*.

Ovviamente, in tutti e tre i metodi, se si trova che nessuna mossa porta ad una posizione presente nella tabella d'apertura, la scelta della mossa da giocare viene effettuata tramite il normale algoritmo Alpha-Beta.
Il primo metodo esegue semplicemente una scelta casuale fra le posizioni che trova nella tabella d'apertura (in sostanza è molto simile al libro standard di GnuChess, che sceglie casualmente tra le varianti

che il libro di apertura gli propone), mentre il secondo e il terzo usano le informazioni della tabella hash nel tentativo di capire quale alternativa sia la migliore. Analizziamo pregi e difetti di questi tre metodi.
Il primo provoca una grande varietà di gioco del programma in fase di apertura, dato che non scarta a priori nessuna variante, ma le considera tutte egualmente giocabili; questo d'altra parte può portare facilmente il programma in posizioni deboli.
Il secondo tende a privilegiare le varianti più popolari, cioè che ricorrono più spesso nel database, a scapito di varianti interessanti, ma poco giocate, magari perché scoperte solo di recente.
Il terzo privilegia, al contrario, le varianti con le migliori percentuali a prescindere dalla frequenza con cui sono state giocate. Ciò porta il programma a delle scelte discutibili: ad esempio, tra una posizione giocata 200 volte con il 70% di vittorie, ed una giocata solo 2 volte con il 100% di vittorie, questo metodo sceglie invariabilmente la seconda. Intuitivamente la prima posizione, anche se ha una percentuale inferiore, ci sembra preferibile, in quanto è stata messa alla prova in un campione di partite più attendibile. Le vittorie ottenute nelle due partite della seconda posizione potrebbero essere dovute, infatti, a errori intervenuti nel prosieguo della partita, e quindi non essere necessariamente significative ai fini della valutazione della posizione in esame.
Il problema della affidabilità del campione di partite, in cui la posizione da valutare è stata raggiunta, si presenta in maniera meno grave anche nel primo e secondo metodo. Nel primo metodo si rischia di seguire varianti poco sperimentate. Nel secondo il problema si presenta quando il programma si trova a dover scegliere tra posizioni tutte poco giocate, e quindi a decidere in base a dati non significativi.
Per evitare questo problema si è limitata la scelta tra le posizioni che hanno almeno 20 occorrenze nel database, e, solo per il terzo metodo, almeno il 10% delle occorrenze dell'ultima posizione raggiunta in partita. Quest'ultimo vincolo è utile quando si esaminano posizioni con molte occorrenze nel database: in questi casi la soglia di occorrenze necessarie per considerare attendibili lo score viene innalzata da 20 partite al 10% delle occorrenze della posizione corrente. Ad esempio se la posizione corrente ha 20.000 occorrenze nel database, i seguiti che portano a posizioni con meno di 2.000 occorrenze non vengono presi in considerazione.
Le formule della funzione $score(p)$ per i tre metodi diventano quindi:

1.

$$score(p) = \begin{cases} k & \text{se } p \in \mathrm{H} \wedge N(p) \geq 20 \\ 0 & \text{altrimenti} \end{cases}$$

2.

$$score(p) = \begin{cases} W(p) + \frac{1}{2}D(p) & \text{se } p \in \mathrm{H} \wedge N(p) \geq 20 \\ 0 & \text{altrimenti} \end{cases}$$

3.

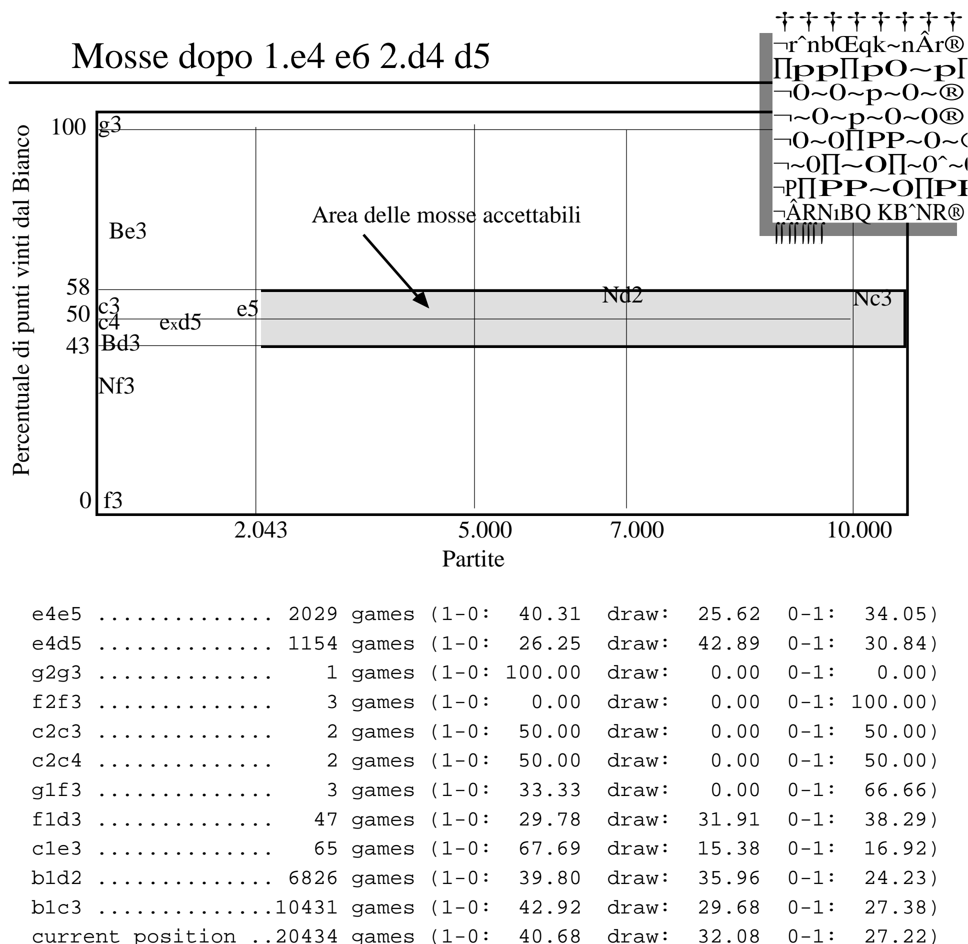
$$score(p) = \begin{cases} \frac{W(p) + ½D(p)}{N(p)} & \text{se } p \in \mathrm{H} \wedge N(p) \geq 20 \wedge N(p) \geq N(p')/10 \\ 0 & \text{altrimenti} \end{cases}$$

dove *p'* è la posizione alla radice dell'albero di gioco, cioè la posizione raggiunta sulla scacchiera prima che inizi la scelta della mossa, mentre *N(p)* restituisce il numero di partite presenti nel database in cui occorre la posizione *p*.

Inoltre, per aumentare la varietà del gioco ottenuto con il terzo criterio, si è introdotta una variabile casuale uniformemente distribuita nell'intervallo [0,0.15] da aggiungere a *score(p)*. In questo modo si evita che il giocatore artificiale giochi in apertura sempre le stesse mosse nelle stesse posizioni.

Nella figura 4.2 si può vedere un esempio tipico dell'uso dell'applicazione del terzo criterio. Nella tabella sono riportati su ciascuna riga: una mossa che porta ad una posizione che compare nella tabella d'apertura, il numero di partite in cui la posizione si è verificata e le percentuali dei risultati finali di queste partite (nell'ordine: vittoria per il Bianco, patta, vittoria per il Nero). Nell'ultima riga sono riportati i dati relativi alla posizione corrente sulla scacchiera, illustrata nel diagramma in alto a destra. Nel grafico ciascuna mossa è localizzata in base al proprio numero di occorrenze nel database (asse delle ascisse) e dalla percentuale di punti vinti dal Bianco (asse delle ordinate). L'area delle mosse accettabili si ottiene eliminando le posizioni con meno del 10% di occorrenze della posizione corrente (in questo caso il 10% di 20.434) ed eliminando le posizioni con una percentuale di 15 o più punti inferiore rispetto alla migliore mossa non ancora scartata (in questo caso Nd2 con 57,78% di punti vinti dal Bianco).

Come si può osservare tra le mosse possibili vengono scartate quelle con un numero troppo limitato di occorrenze nel database, anche se i risultati, in termini di percentuale di punti vinti, sono migliori delle altre.




```
e4e5 .............. 2029 games (1-0:  40.31  draw:  25.62  0-1:  34.05)
e4d5 .............. 1154 games (1-0:  26.25  draw:  42.89  0-1:  30.84)
g2g3 ..............    1 games (1-0: 100.00  draw:   0.00  0-1:   0.00)
f2f3 ..............    3 games (1-0:   0.00  draw:   0.00  0-1: 100.00)
c2c3 ..............    2 games (1-0:  50.00  draw:   0.00  0-1:  50.00)
c2c4 ..............    2 games (1-0:  50.00  draw:   0.00  0-1:  50.00)
g1f3 ..............    3 games (1-0:  33.33  draw:   0.00  0-1:  66.66)
f1d3 ..............   47 games (1-0:  29.78  draw:  31.91  0-1:  38.29)
c1e3 ..............   65 games (1-0:  67.69  draw:  15.38  0-1:  16.92)
b1d2 .............. 6826 games (1-0:  39.80  draw:  35.96  0-1:  24.23)
b1c3 ..............10431 games (1-0:  42.92  draw:  29.68  0-1:  27.38)
current position ..20434 games (1-0:  40.68  draw:  32.08  0-1:  27.22)
```

Figura 4.2. Rappresentazione grafica e testuale dei dati nella tabella di apertura per le posizioni raggiungibili con una mossa del bianco nella posizione del diagramma.

4.5 Sperimentazione

Per misurare la validità del procedimento descritto nei paragrafi precedenti, si sono effettuati dei test, in particolare dei match, tra una versione modificata di GnuChess 4.0, che fa uso di una tabella di apertura, e GnuChess 4.0 originale con e senza libro di apertura. La tabella di apertura è stata costruita con le posizioni delle partite contenute nel database Deja Vu (fig. 4.3).

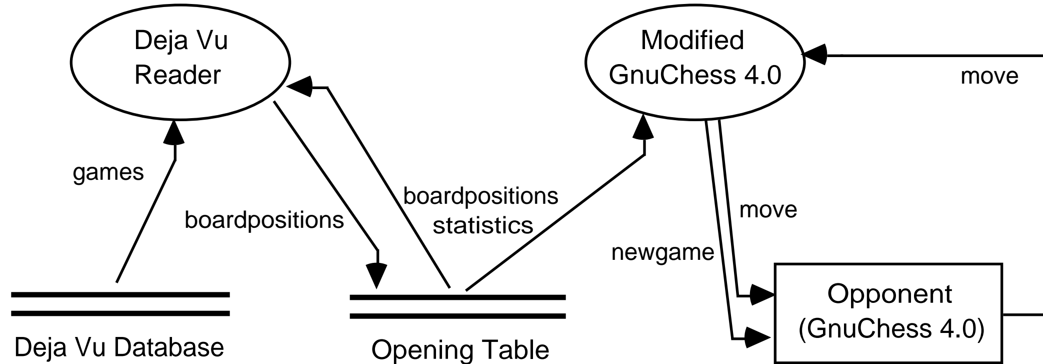


Figura 4.3. Diagramma del flusso dei dati per l'esecuzione dei test.

4.5.1 Generazione delle tabelle di apertura

Date le notevoli dimensioni che la tabella avrebbe dovuto avere per contenere i dati relativi a tutte le 350.000 partite presenti nel database, si è deciso di effettuare match tematici su due aperture: la difesa francese (codici ECO da C00 a C19) e la est-indiana (da E60 a E99). In questo modo sono state generate due diverse tabelle ciascuna di dimensioni tali da poter essere interamente mantenuta in memoria principale durante il test e così limitare l'overhead di accesso alla tabella, anziché accedere ogni volta alla tabella in un file su disco.
Data la disponibilità di memoria RAM per l'esperimento (24 Mb) la tabella di apertura è stata dimensionata a 10 Mb, che consentono la memorizzazione di 2^{20} = 1.048.576 entry. Ciascuna tabella è divisa in due sottotabelle (di 2^{19} entry ciascuna) contenenti separatamente le posizioni per ciascun colore. Nelle tabelle 4.1 e 4.2 sono riportati alcuni dati sulle tabelle hash generate. Le tabelle 4.1 e 4.2 mostrano nelle colonne il numero totale di entry di cui sono composte, comprese le entry vuote, il numero di entry occupate, il numero totale di posizioni inserite (conteggiando più volte le posizioni che si ripetono in più partite) e il numero di posizioni che hanno più di 20 occorrenze nel database. Nelle righe delle tabelle sono divisi i dati relativi alle tabelle di apertura che contengono solo posizioni con mossa al Bianco o al Nero. È da osservare come sia ridotto il numero di posizioni con più di 20 occorrenze nel database, nonostante la considerevole quantità di partite inserite. Questa scelta riduce drasticamente le posizioni contenute effettivamente nel libro di apertura (cioè le posizioni che vengono giocate senza effettuare alcuna ricerca dell'albero di gioco),

tuttavia nel prossimo capitolo vedremo come utilizzare anche le informazioni riguardanti le altre posizioni.

Difesa Francese 22.174 partite	**entry totali**	**entry occupate**	**totale posizioni inserite**	**posizioni con più di 20 occorrenze**
Bianco	524.288	433.804	673.141	913
Nero	524.288	430.243	657.694	913
Totale	1.048.576	864.047	1.330.835	1.826

Tabella 4.1. Dati statistici riguardanti la tabella d'apertura sulla Difesa Francese.

Difesa Est-Indiana 26.816 partite	**entry totali**	**entry occupate**	**totale posizioni inserite**	**posizioni con più di 20 occorrenze**
Bianco	524.288	493.044	811.814	1.156
Nero	524.288	490.273	796.615	1.164
Totale	1.048.576	989.317	1.608.429	2.320

Tabella 4.2. Dati statistici riguardanti la tabella d'apertura sulla Difesa Est-Indiana.

Nella tabella 4.3 sono riportati i dati approssimativi di una ipotetica tabella hash ricavata da tutte le partite del database Deja Vu, calcolati in base alle due tabelle precedenti.

Tutte le aperture 353.000 partite	**entry totali**	**entry occupate**	**totale posizioni inserite**	**posizioni con più di 20 occorrenze**
Bianco	8.388.608	7.000.000	10.500.000	15.000
Nero	8.388.608	7.000.000	10.500.000	15.000
Totale	16.777.216	14.000.000	21.000.000	30.000

Tabella 4.3. Dati statistici riguardanti una tabella d'apertura ricavata da tutte le 353.000 partite di Deja Vu (dati approssimativi stimati).

Da questa stima si ricava che occorrerebbero 160 Mb per la tabella contenente tutte le partite del database. Questo valore si può ridurre a 80 Mb se la struttura `tableEntry` viene così ridefinita:

```
struct tableEntry
{
     unsigned long    hashbd; // codice di controllo
     unsigned char    score;  // punteggio
}
```

dove `score` è un valore tra 0 a 255 ricavato con una delle funzioni *score*(*p*), definite nel paragrafo 4.4, applicata alla posizione relativa a ciascuna entry. Questa struttura consente di risparmiare il 50% di memoria, ma è più rigida, in quanto non è più possibile applicare a tempo di esecuzione del programma di gioco diversi criteri di scelta della mossa, occorre invece definire il criterio di scelta al momento di creare la tabella. Per questo motivo, durante le sperimentazioni si è sempre usata la struttura `tableEntry` come è stata definita nel paragrafo 4.3.

4.5.2 Condizioni dei test

Come detto, i test sono costituiti da match tra due diverse versioni di GnuChess 4.0. Per i match si è scelta la distanza delle 20 partite, ogni giocatore ne gioca 10 con il Bianco e 10 con il Nero. In ogni partita i contendenti hanno a disposizione 25 minuti di riflessione ciascuno per effettuare 50 mosse, quindi una media di 30 secondi per mossa. Questi valori sono considerati un buon compromesso tra attendibilità del risultato del test e risparmio di tempo di elaborazione. Inoltre questi valori ricorrono spesso nei test per giocatori artificiali, tra gli altri sono stati usati anche per i test descritti in [Sch86], [BEGC90], [Toz93], [San93], [Cia94b].
Dato che le modifiche apportate a GnuChess riguardano solo le fasi di apertura, non è necessario, ai fini dei test, far giocare interamente le partite fino a che uno dei due giocatori non prenda matto. Perciò le partite che dopo 50 mosse per parte non sono ancora terminate vengono interrotte. Il risultato di queste partite viene aggiudicato in base allo score attribuito alla posizione finale, tramite un'apposita analisi di un minuto effettuata da GnuChess originale (vedi fig. 4.4).

Inoltre è previsto, per abbreviare i tempi dei test, che un giocatore abbandoni quando, secondo la propria valutazione, lo score della posizione corrente sia sotto i -1000 punti. Questo non modifica l'esito del test: se si verifica questa condizione la sconfitta è assicurata, è solo questione di tempo.

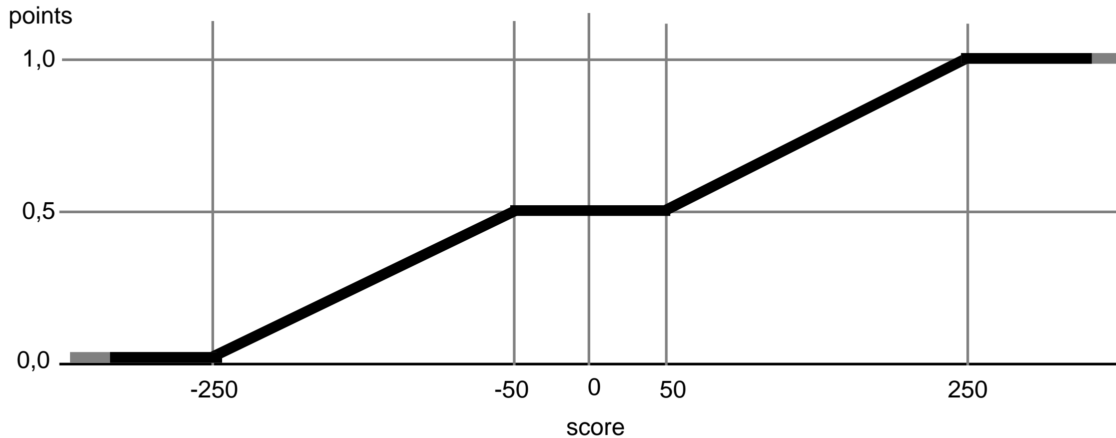


Figura 4.4. Grafico della funzione utilizzata per stabilire il risultato delle partite interrotte alla 50ª mossa.

Al termine di ogni partita il risultato r (r=punti da assegnare al Bianco, i punti da assegnare al Nero sono 1-r) viene così determinato:

r=1 — se il Bianco dà matto entro le 50 mosse, oppure il Nero abbandona, oppure $score > 250$;

r=0 — se il Nero dà matto entro le 50 mosse, oppure il Bianco abbandona, oppure $score < -250$;

r=0,5 — se la partita è patta (casi di triplice ripetizione o stallo), oppure $-50 < score < 50$;

$r=(score+150)/400$ — se la partita è stata interrotta e $50 \leq score \leq 250$;

$r=(250+score)/400$ — se la partita è stata interrotta e $-250 \leq score \leq -50$;

Le partite vengono giocate da due processi avversari che vengono mandati in esecuzione in interleaving sulla stessa macchina (un

PowerMac 7100/66 con 24 Mb di RAM). I due processi comunicano tra loro tramite il meccanismo degli AppleEvents [InsMac]. Lavorando su una architettura uni-processor, le comunicazioni devono essere tutte a rendez-vous esteso. Nel caso di una comunicazione asincrona o a rendez-vous stretto un processo potrebbe togliere del tempo di elaborazione (e quindi di riflessione) all'avversario, dopo avergli inviato la propria mossa; questo potrebbe alterare notevolmente il tempo sfruttato dai due processi, e di conseguenza anche il risultato della partita. I due processi si possono distinguere in processo cliente (la versione modificata di GnuChess 4.0 con tabella delle aperture) che ha l'iniziativa della comunicazione, e in processo servente (GnuChess 4.0, modificato solo per gestire le comunicazioni con il processo cliente) che attende e soddisfa le richieste del cliente (figg. 4.5 e 4.6). Le condizioni dei test ora descritte valgono, per quanto non diversamente specificati, anche per i test cui faranno riferimento i prossimi capitoli.

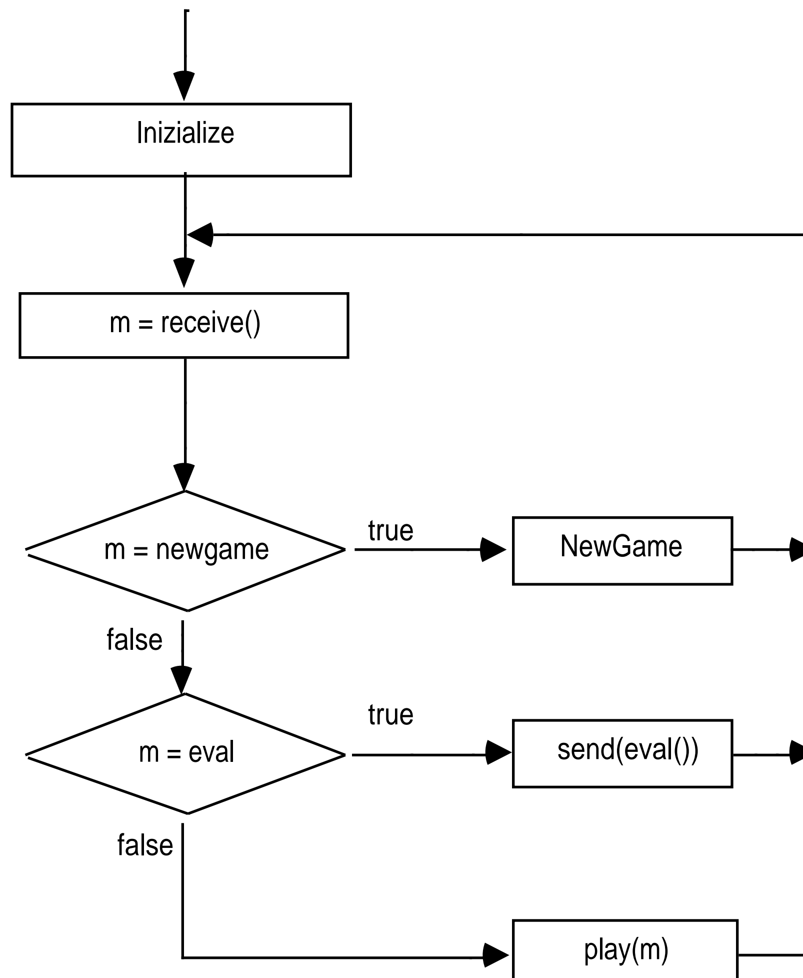


Figura 4.5. Ciclo di base di GnuChess versione Server.

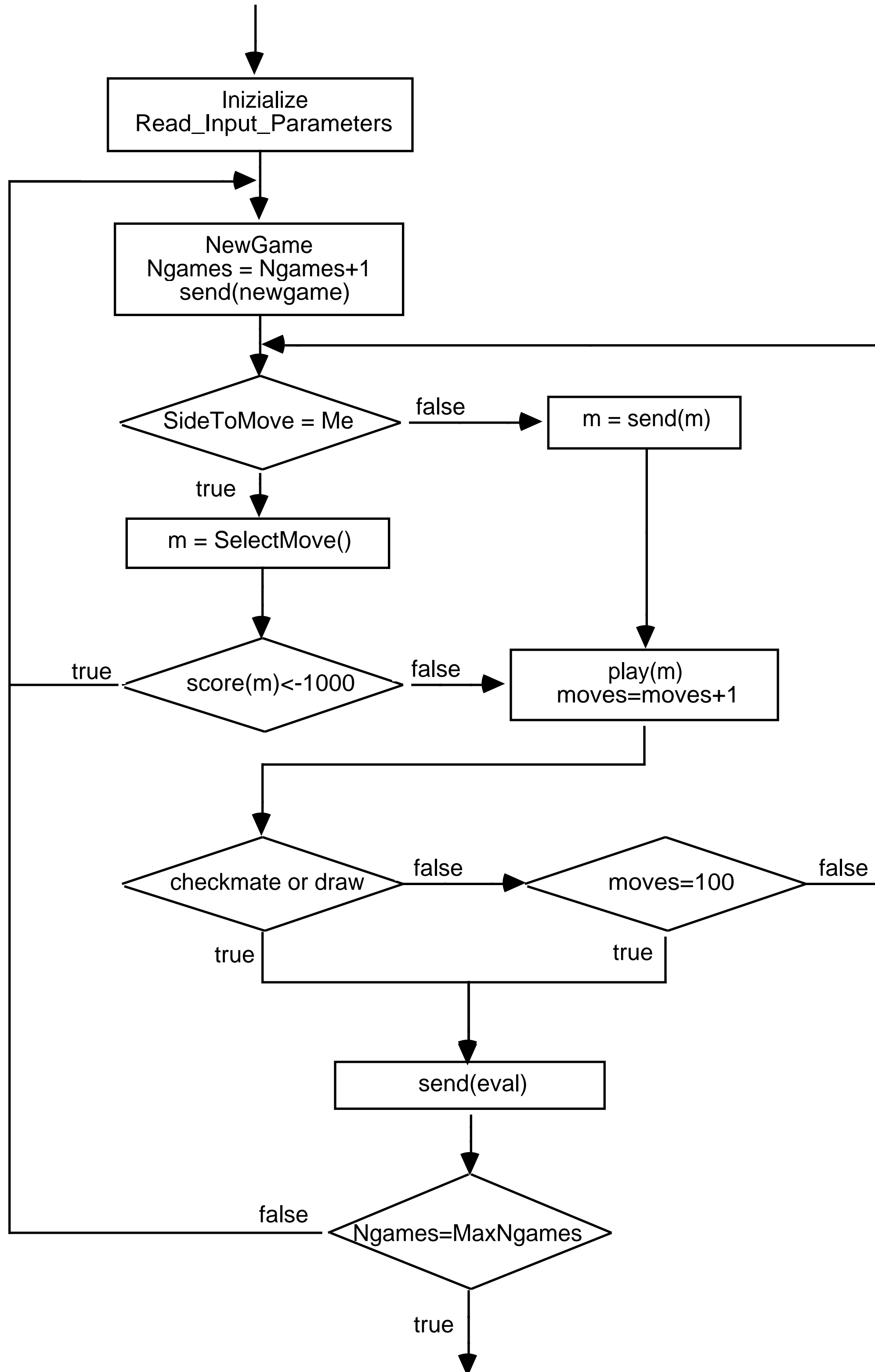


Figura 4.6. Ciclo di base di GnuChess versione Client.

Per verificare che GnuChess 4.0 è un avversario attendibile per la sperimentazione, cioè che non privilegia in modo significativo un colore rispetto all'altro, sono stati eseguiti dei test preliminari. Questi test sono costituiti da tre match giocati da GnuChess contro se stesso. In un match è stato utilizzato tutto il libro di apertura standard di GnuChess, mentre negli altri due sono state usate solo linee di apertura della Difesa Francese e della Difesa Est-Indiana. I risultati del test sono mostrati dalla tabella 4.4.
Questi test preliminari mostrano come il vantaggio che GnuChess dà al Bianco rispecchia abbastanza fedelmente i risultati teorici, che sono stati calcolati in base ai dati statistici di migliaia di partite di torneo tra giocatori umani (vedi in appendice B la tabella B.3). Perciò GnuChess può essere considerato un avversario valido per la sperimentazione.

Apertura	**Giocatori**	**Rilevato**	**Teorico**
Francese	GnuChess4.0-GnuChess4.0	11,50 - 8,50	11,40 - 8,60
Est-Indiana	GnuChess4.0-GnuChess4.0	12,10 - 7,90	11,40 - 8,60
Tutte	GnuChess4.0-GnuChess4.0	11,20 - 8,80	11,20 - 8,80

Tabella 4.4. Match di GnuChess 4.0 contro se stesso.

4.5.3 Risultati

I test sono stati condotti tra GnuChess 4.0, con e senza libro di aperture, e versioni modificate di GnuChess stesso con l'introduzione della tabella di apertura e con uno dei criteri di scelta delle mosse di aperture. Le tre versioni modificate di GnuChess così ottenute sono state chiamate OPTA 1 (OPening TAble), OPTA 2 e OPTA 3, che usano rispettivamente i criteri 1, 2 e 3 definiti nel paragrafo 4.4.
Le tabella 4.5 e 4.6 mostrano i risultati dei test.

Apertura Francese	**GnuChess 4.0**	**GnuChess 4.0 c.l.**
OPTA 1	10,09 - 9,91	7,30 - 12,70
OPTA 2	11,97 - 8,03	11,12 - 8,88
OPTA 3	11,00 - 9,00	9,59 - 10,41

Tabella 4.5. Risultati test sulla Difesa Francese.

Apertura Est-Indiana	**GnuChess 4.0**	**GnuChess 4.0 c.l.**
OPTA 1	11,77 - 8,23	9,73 - 10,27
OPTA 2	11,85 - 8,15	10,50 - 9,50
OPTA 3	10,88 - 9,12	9,96 - 10,04

Tabella 4.6. Risultati test sulla Difesa Est-Indiana.

Innanzi tutto si può osservare che i test condotti sulle due aperture scelte, Difesa Francese e Difesa Est-Indiana, hanno dato risultati abbastanza omogenei. L'unico giocatore a dare un risultato sensibilmente diverso per le due aperture è OPTA 1, che ha conseguito mediamente due punti in più nel match sulla Est-Indiana rispetto a quello sulla Francese. Questo fatto può essere giustificato dalla forte componente casuale presente nella definizione del criterio 1, e forse dalla minore presenza di mosse deboli nelle partite del database riguardanti la Est-Indiana. OPTA 2 e OPTA 3 presentano, invece, scarti minimi tra i risultati dei due match.
Come era logico aspettarsi i migliori risultati sono stati conseguiti contro GnuChess senza libro di aperture, contro il quale tutti i giocatori in esame hanno vinto. Nel match con libro di aperture GnuChess 4.0 recupera mediamente 1÷1,5 punti sui suoi avversari.
Nel complesso OPTA 2 risulta essere il giocatore più forte e OPTA 1 il più debole. È da tenere presente, però, che OPTA 2, per come è definito, ha una scarsa varietà di gioco in apertura, per cui è possibile che abbia ottenuto un certo vantaggio dall'insistere maggiormente su alcuni punti deboli di GnuChess, rispetto a OPTA 3 che varia maggiormente il gioco tra una partita e l'altra. Questa considerazione è confermata da un match giocato da OPTA 2 contro OPTA 3, in cui quest'ultimo risulta vincente per 12,38 a 7,62.

Complessivamente, per OPTA 2 e OPTA 3, si può dire che si è ottenuto un vantaggio nei confronti di GnuChess senza libro e che si è raggiunta all'incirca la forza di gioco di GnuChess con il proprio libro di aperture standard.

Capitolo 5

Euristiche di preordinamento delle mosse

Come si è visto nel capitolo precedente, solo una piccola parte dei dati presenti nel database di partite può essere usata efficacemente per la creazione di un libro di aperture in maniera del tutto automatica. Ciò non significa che il resto dei dati presenti nel database non siano utilizzabili per migliorare la qualità del gioco di un giocatore artificiale. Una tecnica per cui questi dati potrebbero essere molto utili è il preordinamento delle mosse durante la visita dell'albero di gioco. È noto che l'algoritmo AlphaBeta raggiunge la massima efficienza quando visita un albero già ordinato, cioè dove vengono visitati prima i nodi con valore minimax più alto, massimizzando così il numero di tagli effettuati dall'algoritmo. Per questo in tutti i programmi di gioco che fanno uso dell'algoritmo AlphaBeta, e relative varianti, sono presenti varie euristiche per cercare di ordinare correttamente i nodi intermedi dell'albero di gioco.
In questo capitolo si propongono due tecniche per cercare di ricavare da database di partite informazioni utili ai fini del preordinamento delle mosse.

5.1 Prima tecnica

Supponiamo che il nostro giocatore artificiale usi la tabella di apertura con il terzo criterio di scelta definito nel capitolo precedente. Quando, durante la partita, non trova più mosse che portano a posizioni con un numero di occorrenze nel database sufficiente per essere giocate automaticamente, attiva l'algoritmo AlphaBeta. Tuttavia, molto proba-

bilmente, nella tabella di apertura ci sono dati statistici relativi ad alcune posizioni che ora l'algoritmo AlphaBeta andrà ad esaminare. Questi dati potrebbero suggerire un criterio di ordinamento per le posizioni durante la costruzione dell'albero di gioco. In sostanza il comportamento complessivo del giocatore artificiale potrebbe essere il seguente: finché trova posizioni con un numero sufficiente di occorrenze prende per buoni i relativi dati statistici per ricavarne una valutazione, altrimenti questa valutazione viene presa in considerazione solo come suggerimento per l'ordinamento delle mosse.

5.1.1 Implementazione

L'implementazione di questa tecnica si ricava direttamente dall'estensione dell'uso della tabella di apertura anche alle posizioni con poche occorrenze nel database (meno di 20, considerando la soglia stabilita nel capitolo precedente). Le informazioni relative a queste mosse non servono per stabilire immediatamente quale sia la mossa da giocare, ma solo per ordinare i nodi visitati dall'algoritmo AlphaBeta. I nodi (posizioni) vengono ordinati in ordine decrescente in base alla funzione *s_score(p)* così definita:

$$s_score(p) = \begin{cases} score(p) + M & \text{se } p \in \mathrm{H} \\ sorting_score(m(p', p)) & \text{altrimenti} \end{cases}$$

dove *score(p)* è definita in uno dei tre modi descritti nel paragrafo 4.4, e *sorting_score(m(p',p))* riassume tutte le euristiche di preordinamento delle mosse presenti in GnuChess 4.0 (vedi appendice A.2) applicate alla mossa *m(p',p)* che porta dalla posizione *p'* alla posizione *p*. La costante *M* è tale che

$$M > sorting_score(m(p,q)) \qquad \forall p,q \text{ ,}$$

in modo da dare sempre la precedenza alle posizioni presenti nel database rispetto alle altre.
Per limitare l'overhead dovuto all'accesso alla tabella di apertura durante l'esecuzione dell'algoritmo AlphaBeta la funzione *s_score(p)* viene calcolata come sopra descritto solo quando il programma è appena uscito dal libro di aperture oppure se durante l'analisi per la

mossa precedente sono state incontrate posizioni presenti nella tabella di apertura, altrimenti la funzione *s_score(p)* viene sostituita da *sorting_score(m(p',p))*, evitando così di accedere alla tabella di apertura per controllare se la posizione *p* vi è presente.

5.1.2 Sperimentazione

La sperimentazione è stata condotta con match di 20 partite sulla difesa francese e sulla est-indiana. La versione fatta giocare contro GnuChess 4.0 è OPTA 3.1, derivata da OPTA 3 (vedi capitolo precedente) con l'aggiunta della euristica di ordinamento delle mosse ora descritta. Inoltre è stato effettuato un match tra OPTA 3.1 e OPTA 3.

Risultati apertura francese	
OPTA 3.1 - GnuChess 4.0	12,92 - 7,08
OPTA 3.1 - GnuChess 4.0 con libro	11,35 - 8,65
OPTA 3.1 - OPTA 3	13,68 - 6,32

Tabella 5.1. Test di OPTA 3.1 sull'apertura Francese.

Risultati con apertura est - indiana	
OPTA 3.1 - GnuChess 4.0	12,68 - 7,32
OPTA 3.1 - GnuChess 4.0 con libro	11,30 - 8,70
OPTA 3.1 - OPTA 3	10,82 - 9,18

Tabella 5.2. Test di OPTA 3.1 sull'apertura Est-Indiana.

Il miglioramento di prestazioni di OPTA 3.1 rispetto a OPTA 3, e il risultato dei loro scontri diretti (tabelle 5.1 e 5.2), conferma l'intuizione che le mosse suggerite dal database sono in genere buone, capaci di provocare tagli dell'albero di gioco.

Che le mosse suggerite dal database provochino tagli consistenti all'albero di gioco è confermato anche dalla tabella 5.3. In questa tabella sono riportati il numero di nodi necessari a GnuChess 4.0 e a OPTA 3.1 per completare la visita dell'albero di gioco con la modalità dell'Iterative Deepening a diverse profondità *d*. I dati riportati sono i valori medi ricavati dalle ricerche compiute su 100 posizioni di mediogioco riguardanti l'apertura francese estratte da [Nik89]. I tagli dell'albero provo-

cati da OPTA 3.1 consentono di risparmiare circa il 25% di tempo a profondità 4 e 5, e quasi il 30% al livello di profondità 6. Per dare un'idea dell'aumento di prestazioni ottenuto, si può dire che GnuChess per raggiungere la stessa profondità di ricerca di OPTA 3.1 nello stesso tempo ha bisogno di un hardware circa del 40% più veloce.
Naturalmente questo guadagno di prestazioni si ha solo nelle posizioni prossime all'apertura, mentre verso il finale di partita i due programmi si equivalgono, dato che OPTA 3.1 non è più in grado di trovare nella tabella di apertura posizioni che lo possano guidare durante l'analisi.

d	GnuChess 4.0	OPTA 3.1	diff. %
4	44.519	33.552	- 24,46
5	174.315	130.597	- 25,08
6	1.176.813	833.687	- 29,16

Tabella 5.3. Confronto tra il numero di nodi visitati da GnuChess 4.0 e da OPTA 3.1

5.1.3 Ulteriore miglioramento tramite feedback sul database

Per estendere la conoscenza scacchistica in apertura del nostro giocatore artificiale è possibile arricchire i dati della tabella di apertura con quelli provenienti dall'esperienza di gioco del giocatore stesso. Tenere conto dell'esperienza comporta vantaggi e svantaggi. Il principale vantaggio, oltre naturalmente all'ampliamento delle posizioni nel libro di aperture, è che le varianti di apertura scelte dal programma in base alla propria esperienza saranno quelle che gli sono più congeniali, evitando il classico problema degli errori subito dopo l'uscita dal libro. Il difetto è che queste varianti non si adatteranno solo alle caratteristiche del programma che apprende, ma anche alle debolezze del giocatore che lo allena. Il pericolo è che varianti che risultano vincenti contro l'allenatore risultino poi perdenti contro giocatori più forti dell'allenatore, o comunque con caratteristiche diverse. L'ideale sarebbe poter allenare il programma contro diversi giocatori, tutti di alto livello. Questo purtroppo non è sempre possibile.
Nel nostro caso il programma allenatore è OPTA 3.1, mentre il programma che apprende, chiamato OPTA FB, funziona sostanzialmente

nello stesso modo di OPTA 3.1, con la differenza che al termine di ogni partita, in base al risultato, la tabella di apertura viene modificata in tutte le entry corrispondenti alle posizioni che si sono susseguite durante la partita stessa. Affinché le nuove informazioni inserite nella tabella di apertura diventino rilevanti, in confronto alle migliaia di partite già inserite, occorre un considerevole numero di partite di allenamento. Per motivi di tempo le partite sono state giocate a cadenza molto rapida (2 minuti per 50 mosse), e sono fatte valere come 5 partite provenienti dal database. Per le partite non concluse entro la 50ª mossa viene effettuata una valutazione della posizione finale raggiunta. Dato che le partite valgono come 5 partite del database, per i valori che non attribuiscono nettamente la vittoria ad uno dei due giocatori, è possibile assegnare alla partita un risultato misto. La tabella di apertura viene quindi modificata in base al risultato di questa valutazione secondo la funzione illustrata in figura 5.1.

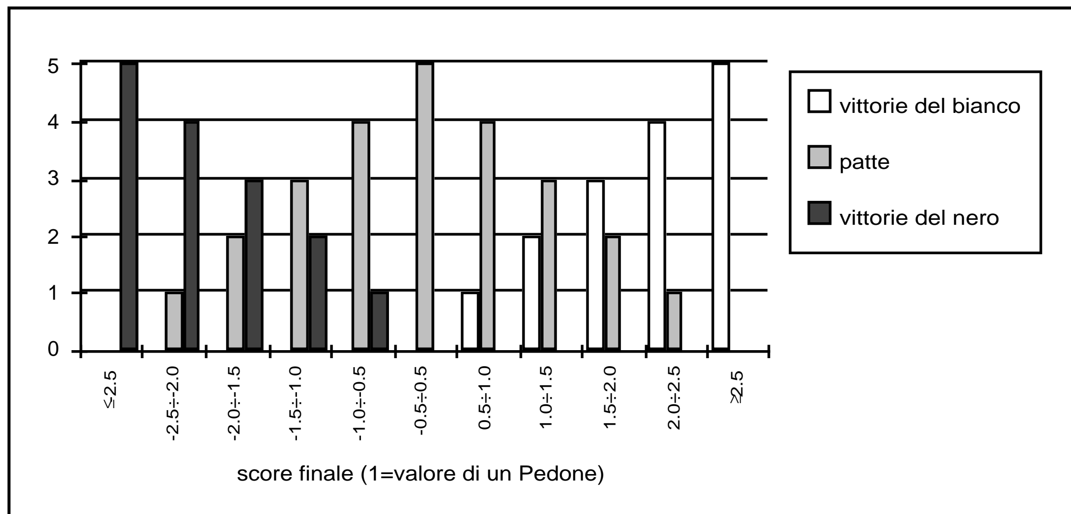


Figura 5.1. Grafico della funzione che determina il risultato della partita in base allo score assegnato dalla valutazione.

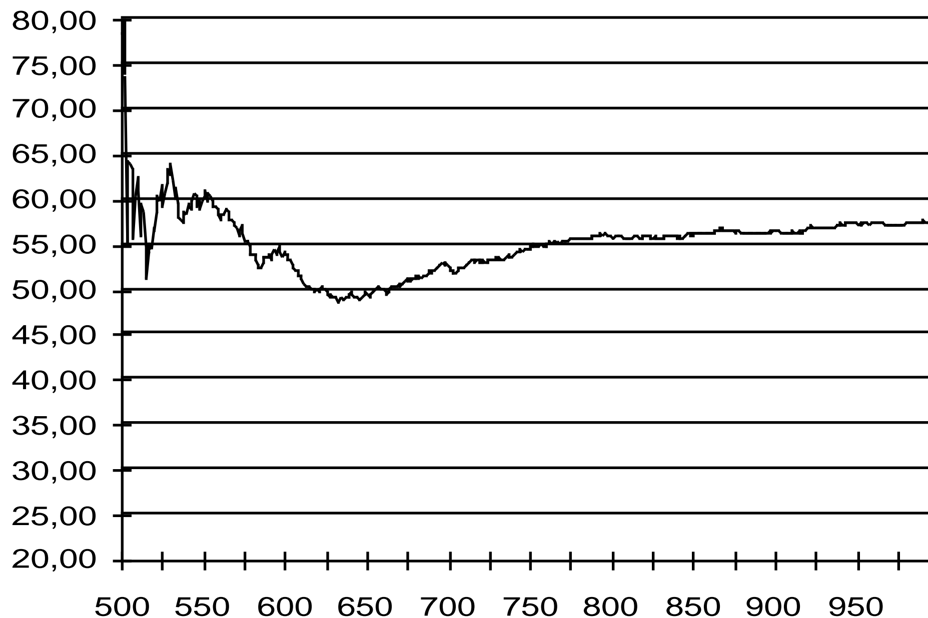


Figura 5.2. Andamento della percentuale di punti vinti da OPTA FB.

Sono state giocate 1000 partite di allenamento. Nelle prime 500 OPTA FB non ha sfruttato l'esperienza accumulata, in modo da non restringere troppo presto l'apprendimento a poche varianti. Nella figura 5.2 si può vedere l'andamento della percentuale di punti vinti da OPTA FB su OPTA 3.1 nelle seconde 500 partite di allenamento. Dopo una oscillazione iniziale si può osservare come la percentuale di punti vinti da OPTA FB salga con una certa continuità stabilizzandosi intorno al 58%. Sono stati effettuati anche due test (con match sulle 20 partite, con la consueta cadenza di 25 minuti per 50 mosse) dopo 750 e 1000 partite (vedi tabella 5.4). I risultati mostrano l'efficacia del procedimento applicato. È da notare che nonostante l'apprendimento sia avvenuto con partite lampo, questo ha avuto effetti molto positivi anche sul gioco a cadenza più lenta. Inoltre l'allenamento contro OPTA 3.1 ha dato i suoi frutti anche nel match contro GnuChess 4.0, dove OPTA FB ha totalizzato 12,22 punti, rispetto agli 11,35 di OPTA 3.1 (tabella 5.1). Comunque, come era logico aspettarsi, il miglioramento nei confronti dell'allenatore OPTA 3.1 è stato assai più marcato che nei confronti di GnuChess 4.0, che avendo un diverso libro di aperture può portare a volte OPTA FB su linee di gioco sulle quali non ha acquisito molta esperienza.

Risultati con apertura francese	
OPTA FB 750 - OPTA 3.1	12,50 - 7,50
OPTA FB 1000 - OPTA 3.1	13,28 - 6,72
OPTA FB 1000 - GnuChess 4.0 con libro	12,22 - 7,88

Tabella 5.4. Match di OPTA FB.

5.2 Seconda Tecnica

Un'altra tecnica per cercare di ottimizzare l'ordinamento delle mosse, funziona nel seguente modo: data la prima posizione in cui il programma si trova a giocare fuori dal proprio libro di aperture, un modo per ordinare le mosse che da questa si possono generare è considerare le mosse più giocate, nel database di partite, nell'ultima posizione della partita trovata nel libro di aperture. Volendo incrementare la quantità di informazioni a disposizione del programma per effettuare l'ordinamento delle mosse si può estendere la profondità delle mosse del database prese in considerazione. Cioè si può tenere conto delle mosse nel database giocate a partire da una determinata posizione, fino alla profondità n fissata, e quindi utilizzare i dati così ricavati per ordinare le mosse dell'albero di gioco fino alla stessa profondità n.
Questa tecnica si basa sul presupposto che una mossa buona per una determinata posizione lo sia anche per una posizione simile. Questo negli Scacchi in genere non è vero, in quanto anche piccole differenze, come ad esempio la casa occupata da un singolo Pedone, possono modificare radicalmente la giusta strategia da intraprendere da parte di un giocatore. Tuttavia questa tecnica sembra essere promettente in quanto generalizza maggiormente, rispetto alla tecnica illustrata nel paragrafo precedente, le conoscenze ricavate dal database di partite, applicandole a posizioni che nel database possono anche non comparire affatto.

5.2.1 Implementazione

Una semplice implementazione di questa tecnica potrebbe consistere nell'analizzare il database di partite appena il giocatore artificiale esce dal libro di apertura per vedere quali siano state le mosse giocate a partire dalla posizione che sta esaminando. Ciò è però irrealizzabile a causa del tempo necessario all'analisi del database (dell'ordine di diversi minuti) inaccettabile per partite con controlli di tempo.
È necessario quindi analizzare preventivamente il database e memorizzare, separatamente per ogni posizione di apertura, le informazioni relative alle mosse giocate a partire da ciascuna posizione. In questo modo il nostro giocatore artificiale è pronto ad utilizzare questa tecnica in qualsiasi posizione il suo avversario lo faccia uscire dal libro di aperture. Per ogni posizione del libro di aperture viene generata una tabella (*HistoryTable*) che associa ad ogni mossa un valore, che sarà poi usato per l'ordinamento (fig. 5.3).

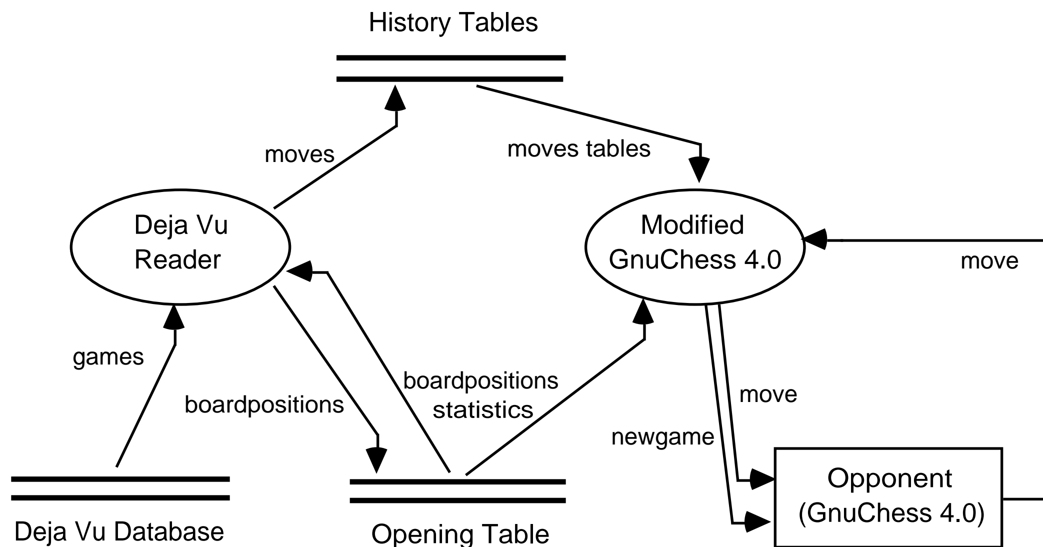


Figura 5.3. Diagramma del flusso dei dati con l'impiego delle *HistoryTable*.

Ogni *HistoryTable* è così definita:

```
unsigned char HistoryTable[64][64];
```

in `HistoryTable[x][y]` è contenuto il valore attribuito alla mossa che corrisponde al movimento del pezzo dalla casa `x` alla casa `y`. Nella

HistoryTable non si fa distinzione tra mosse del Bianco e mosse del Nero, e neppure sul tipo del pezzo mosso. Queste semplificazioni sono accettabili, dato che stiamo trattando posizioni della fase di apertura e abbastanza simili tra loro, per cui l'indicazione della casa di partenza e la casa di arrivo è sufficiente a individuare senza equivoci la mossa a cui la *HistoryTable* si riferisce.
Tra gli svariati criteri che si potrebbero definire per generare le *HistoryTable*, e quindi ordinare le mosse in fase di esecuzione, è stato scelto il seguente: le mosse vengono ordinate primariamente rispetto alla distanza minima (dalla posizione di partenza della *HistoryTable*) in cui sono state giocate, e secondariamente rispetto al numero di occorrenze che hanno nel database. In questo modo durante l'analisi si dà la precedenza alle mosse giocate immediatamente nella posizione di partenza, e, a parità di distanza dalla posizione di partenza, a quelle più frequentemente giocate.
Formalmente, ogni elemento di una *HistoryTable*, relativa alla posizione di apertura *p*, viene così calcolato:

```
HistoryTable[f][t]=64*(4-(int)(d(f,t)/2))+63*O(f,t)/OMAX(d(f,t))
```

`f` e `t` indicano rispettivamente la casa di partenza e di arrivo della mossa, `d(f,t)` restituisce la distanza minima a cui si trova la mossa da `f` a `t` rispetto alla posizione *p*, cioè il minimo numero di semimosse che intercorre tra l'occorrenza della posizione di partenza e l'occorrenza della mossa `(f,t)` nelle partite del database. `O(f,t)` restituisce il numero di partite del database in cui occorre la mossa `(f,t)` dopo l'occorrenza della posizione *p*, mentre `OMAX(d(f,t))` restituisce il numero di occorrenze della mossa, di distanza minima `d(f,t)`, con più occorrenze nel database.
Per il calcolo delle *HistoryTable* vengono prese in considerazione solo le mosse distanti fino a 8 semimosse dalla posizione di partenza, e a tempo di esecuzione questa tecnica non viene utilizzata quando si analizzano mosse distanti più di 8 semimosse dall'ultima posizione di libreria, questo perché maggiore è la profondità, maggiori sono le differenze tra l'ultima posizione di libreria incontrata e la posizione attuale, per cui il presupposto che lo mosse presenti nella *HistoryTable* siano buone anche per la posizione attuale diventa sempre più debole.
Un esempio dei dati contenuti in una *HistoryTable* ricavata dalle partite del database è illustrato nella tabella 5.5, relativamente alla posizione del diagramma in figura 5.4.

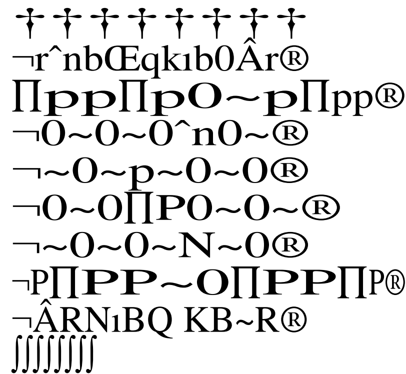


Figura 5.4. Una posizione di partenza per costruire una *HistoryTable*.
Mossa al Bianco.

Si può osservare che le *HistoryTable* risultano essere delle matrici sparse: in primo luogo perché le regole del gioco rendono impossibili

Mosse del Bianco	**punteggio**	**Mosse del Nero**	**punteggio**
f1d3	255	f8d6	255
c1g5	219	f8e7	236
b1c3	212	c8g4	207
h2h3	206	c7c5	200
c2c4	200	f8b4	198
f1e2	197	h7h6	197
c1e3	195	b8c6	195
c1f4, a2a3	194	c7c6, c8e6, c8f5	194
00	191	00	191
d1e2	134	d8e7	133
c2c3	133	c5c4	132
c1d2, d4c5	130	g4h5	130
f4g3, g5h4	129	b4d2, b4d6, f5d3, f6g4, f8g5	129
f1e1	127	...	
b1d2, c4d5	92		
e2e7	89		
g5f6	82		
d1d2, f1c4,f1b5, b2b3, e5c6	68		
...			

Tabella 5.5. Le mosse con punteggio più alto presenti nella *HistoryTable* relativa alla posizione del diagramma nella fig. 5.4.

molte mosse (ad es. nessun pezzo può eseguire la mossa a1d2) e poi perché le mosse sperimentate, che sono nel database, sono in genere una piccola parte delle mosse legali. Si può quindi risparmiare molta memoria utilizzando una rappresentazione alternativa per le *HistoryTable*. Per questo per memorizzare le *HistoryTable* su disco sono state utilizzate liste di record così definiti:

```
struct move
{
      unsigned char f;        // casa di partenza
      unsigned char t;        // casa di arrivo
      unsigned char points;   // punteggio
};
```

Tutte le *HistoryTable* sono state registrate nel medesimo file separate da una record vuoto. Un altro file indice permette di accedere alle liste di mosse tramite puntatori al primo file. La struttura ad array per le *HistoryTable* viene usata comunque in memoria principale in quanto permette una maggiore velocità di accesso, dove comunque è necessaria una *HistoryTable* alla volta che viene caricata dal file delle *HistoryTable* all'inizio dell'analisi dell'albero di gioco.
La tecnica delle *HistoryTable* viene usata insieme a quella descritta precedentemente, che fa uso della tabella di apertura, per cui a tempo di esecuzione il punteggio da assegnare alle posizioni, per il loro ordinamento durante l'analisi, è così definito:

$$s_score(p) = \begin{cases} score(p) + M + HistoryTable[f][t] & \text{se } p \in \mathrm{H} \\ sorting_score(m(p',p)) + HistoryTable[f][t] & \text{altrimenti} \end{cases}$$

dove *p'* è la posizione nel nodo padre di *p* e *m(p',p)=(f,t)* è la mossa corrispondente all'arco che va da *p'* a *p*.

5.2.2 Sperimentazione

Il giocatore artificiale testato è basato su OPTA 3.1 (vedi paragrafo 5.1), con in più l'uso delle *HistoryTable*. Il programma così ottenuto è stato chiamato OPHISTA (OPening and HIStory TAbles). Nella tabella 5.6 sono riportati i risultati di alcuni test che misurano la quantità di

nodi visitati a diverse profondità di ricerca da OPTA 3.1 e OPHISTA. Per questi test sono state utilizzate 50 posizioni scelte casualmente tra quelle subito fuori dal libro di aperture. L'efficacia dei tagli prodotti dall'uso delle *HistoryTable* nell'analisi di queste posizioni risulta evidente: si ha in media un guadagno del 34% di nodi visitati, ciò significa che nelle posizioni del test OPTA 3.1 per raggiungere la stessa profondità di analisi, a parità di tempo rispetto a OPHISTA, deve girare su un hardware più veloce di circa il 50%. Rispetto alla analoga tabella 5.3 si può osservare che il risparmio percentuale di nodi visitati che si ha a profondità 6 non aumenta rispetto al risparmio che si ha a profondità 5, ma anzi si ha una leggero peggioramento. Ciò probabilmente è dovuto a quanto si è già detto nel paragrafo precedente: maggiori sono le differenze tra l'ultima posizione di libreria incontrata e la posizione attuale, meno è probabile che le mosse suggerite dalla *HistoryTable* siano buone. Dato che aumentando la profondità di ricerca si trovano posizioni più diverse dalla posizione radice, può darsi che in qualche caso le mosse suggerite dalla *HistoryTable* durante l'analisi a profondità 6 non siano state tra le migliori.

d	**OPTA 3.1**	**OPHISTA**	**diff. %**
4	28.959	19.384	- 33,06
5	127.462	83.463	- 34,52
6	833.268	550.610	- 33,92

Tabella 5.6. Confronto tra il numero di nodi visitati da OPTA 3.1 e OPHISTA.

Risultati con apertura francese	
OPHISTA - GnuChess 4.0	12,53 - 7,47
OPHISTA - OPTA 3.1	10,98 - 9,02

Tabella 5.7 Esperimenti con *HistoryTable*.

Nella tabella 5.7 sono riportati i risultati dei match effettuati da OPHISTA. È da notare che l'introduzione delle *HistoryTable* non ha portato miglioramenti nel match contro GnuChess 4.0 senza libro di aperture. Questo è dovuto al fatto che GnuChess costringe OPHISTA ad uscire troppo presto dal proprio libro di aperture, e quindi ad usare le *HistoryTable* in una fase della partita troppo vicina all'apertura per

portare benefici significativi. Al contrario, le *HistoryTable* hanno dimostrato la loro efficacia nel match contro OPTA 3.1. Qui i due avversari hanno lo stesso libro di aperture e quindi le *HistoryTable* vengono impiegate in una fase più avanzata della partita e il loro uso può risultare determinante per l'esito finale.
In conclusione si può affermare che l'uso di questa tecnica di preordinamento delle mosse è utile soprattutto con avversari con una buona preparazione teorica sulla fase di apertura.

Capitolo 6

Applicazione di tecniche di ML per la funzione di valutazione statica

Le tecniche di apprendimento da database di partite viste finora, pur dando risultati positivi, hanno il limite di una scarsa generalizzazione delle conoscenze acquisite dal database. Queste infatti vengono applicate solo quando si incontrano posizioni che ricorrono nel database o al massimo quando si incontrano posizioni molto simili (nel caso delle *HistoryTable*). Quando il giocatore artificiale si trova a dover giocare in una situazione al di fuori di questo insieme di posizioni, concentrate soprattutto nella fase di apertura, le conoscenze acquisite dal database non vengono utilizzate affatto.
L'ideale sarebbe poter acquisire conoscenze dal database che aiutino il programma di gioco anche in posizioni molto diverse da quelle presenti nel database stesso. Per raggiungere questo obiettivo occorre agire in qualche modo sulla funzione di valutazione statica.
In questo capitolo vengono presentati due diversi approcci a questo problema: l'apprendimento Bayesiano e gli algoritmi genetici.

6.1 Apprendimento Bayesiano

L'apprendimento Bayesiano, come si è già visto nel paragrafo 3.3, è usato per classificare elementi di un database, in base ai loro attributi. L'appartenenza di un elemento x_i del database ad una classe C_j non viene stabilita dall'algoritmo in maniera certa, ma ne viene data la probabilità, usando il teorema di Bayes:

$$p(x_i \in C_j | x_i, \vec{\theta}, \vec{\pi}, J) = \frac{\pi_j p(x_i | x_i \in C_j, \vec{\theta}_j)}{p(x_i | \vec{\theta}, \vec{\pi}, J)} \quad (1)$$

dove J è il numero complessivo delle classi, $\vec{\theta}$ il vettore dei descrittori delle distribuzioni degli attributi per ciascuna classe, e $\vec{\pi}$ il vettore delle probabilità delle classi, per cui si ha $\pi_j = p(x_i \in C_j)$. Da ciò deriva che un elemento può far parte di classi diverse con probabilità diverse. Quindi la probabilità di un elemento appartenente all'insieme delle J classi (al denominatore della (1)) è la somma delle probabilità che ha di appartenere a ciascuna classe, pesata per le probabilità delle classi. Cioè

$$p(x_i | \vec{\theta}, \vec{\pi}, J) = \sum_{j=1}^{J} \pi_j p(x_i | x_i \in C_j, \vec{\theta}_j) \quad (2)$$

Un vantaggio dell'apprendimento Bayesiano, rispetto ad altri algoritmi di classificazione, è la capacità di individuare autonomamente il numero delle classi, e eventualmente una loro gerarchia, senza supervisione, cioè senza che gli sia fornito alcun esempio di elemento appartenente ad una classe (vedi [CKSSTF88]). Comunque questa caratteristica non è necessaria per lo sviluppo di funzioni di valutazione per i giochi a informazione completa.

6.1.1 Apprendimento Bayesiano nei giochi a informazione completa

L'approccio dell'apprendimento Bayesiano per lo sviluppo di una funzione di valutazione per un gioco ad informazione completa è di natura statistica. La funzione di valutazione viene vista come una variabile aleatoria, indicante la probabilità di vittoria di uno dei due contendenti, dipendente da altre variabili aleatorie: i vari fattori di conoscenza della funzione di valutazione. L'idea è di osservare delle posizioni campione di cui si conosce già una valutazione, non assegnata dal programma, e stabilire una relazione tra i valori delle varie euristiche di valutazione e la valutazione nota. Questa relazione verrà poi generalizzata a tutte le posizioni che la funzione di valutazione sarà chiamata a valutare. In pratica la funzione dovrà classificare le posizioni che gli vengono presentate sulla base delle posizioni campione.
Le posizioni campione vengono divise in tre classi:

- W: vincenti per il giocatore che muove;
- D: posizioni di patta;
- L: perdenti per il giocatore che muove.

La classe D può essere trascurata per i giochi in cui le patte non si possono verificare oppure sono molto rare, come ad esempio nel gioco dell'Othello.

Il punteggio atteso in una posizione x per il giocatore che ha la mossa è dato da:

$$\frac{p(x \in W|x,\vec{\theta},\vec{\pi}) + \frac{1}{2} p(x \in D|x,\vec{\theta},\vec{\pi})}{p(x \in W|x,\vec{\theta},\vec{\pi}) + p(x \in D|x,\vec{\theta},\vec{\pi}) + p(x \in L|x,\vec{\theta},\vec{\pi})} \quad (3)$$

Per semplicità consideriamo le posizioni campione come indipendenti tra loro. Per cui le distribuzioni dei valori dei fattori di conoscenza (o attributi della posizione) all'interno delle tre classi possono essere stimate statisticamente con leggi normali. Possiamo scegliere se considerare anche i fattori di conoscenza indipendenti oppure no. Nel primo caso possiamo stimare la distribuzione di ciascun fattore con una gaussiana. Tuttavia è noto che i fattori di conoscenza non sono tra loro indipendenti (ad es. negli Scacchi è probabile che una maggiore mobilità comporti anche un maggiore controllo del centro, o un vantaggio di materiale) per cui sembra più logico stimare le distribuzioni degli attributi con leggi normali multivariate, $N(\vec{\mu}_W, A_W)$, $N(\vec{\mu}_D, A_D)$ e $N(\vec{\mu}_L, A_L)$ rispettivamente per le tre classi definite, anche se questo comporta un maggior costo computazionale. I vettori delle medie $\vec{\mu}_W$, $\vec{\mu}_D$, $\vec{\mu}_L$ sono calcolati con una media empirica degli attributi delle posizioni campione:

$$\mu_{W_i} = \sum_{\forall x \in W} \frac{x_i}{\text{card}(W)}; \quad \mu_{D_i} = \sum_{\forall x \in D} \frac{x_i}{\text{card}(D)}; \quad \mu_{L_i} = \sum_{\forall x \in L} \frac{x_i}{\text{card}(L)} \quad (4)$$

per ogni fattore di conoscenza i di cui il giocatore artificiale che apprende è dotato. Anche gli elementi delle matrici di covarianza A_W, A_D, A_L sono ricavati empiricamente dai dati campione, i rispettivi elementi sono calcolati con le formule:

$$a_{W_{i,j}} = \sum_{\forall x \in W} \frac{x_i x_j}{\mathrm{card}(W)} - \mu_{W_i}\mu_{W_j}; \quad a_{D_{i,j}} = \sum_{\forall x \in D} \frac{x_i x_j}{\mathrm{card}(D)} - \mu_{D_i}\mu_{D_j};$$
$$a_{L_{i,j}} = \sum_{\forall x \in L} \frac{x_i x_j}{\mathrm{card}(L)} - \mu_{L_i}\mu_{L_j} \tag{5}$$

Il vettore delle probabilità a priori viene posto $\vec{\pi}=(1/3,1/3,1/3)^T$. Dalla densità delle leggi normali multivariate (vedi [Bal92]) e dalla (1) si ha che data una posizione x qualsiasi, questa può essere attribuita a ciascuna delle tre classi con le probabilità definite dalle funzioni:

$$f_W(x) = p(x \in W | x, \vec{\theta}, \vec{\pi}) = \frac{\pi_W p(x | x \in W, \vec{\theta}_V)}{p(x|\vec{\theta}, \vec{\pi}, J)} = \frac{\exp\left(-\frac{1}{2}(x-\vec{\mu}_W)^{\mathrm{T}} \mathrm{A}_{\mathrm{W}}^{-1}(x-\vec{\mu}_W)\right)}{3(2\pi)^{\frac{m}{2}} |\mathrm{A}_{\mathrm{W}}|^{\frac{1}{2}} p(x|\vec{\theta}, \vec{\pi}, J)}$$

$$f_D(x) = p(x \in D | x, \vec{\theta}, \vec{\pi}) = \frac{\pi_W p(x | x \in D, \vec{\theta}_D)}{p(x|\vec{\theta}, \vec{\pi}, J)} = \frac{\exp\left(-\frac{1}{2}(x-\vec{\mu}_D)^{\mathrm{T}} \mathrm{A}_{\mathrm{D}}^{-1}(x-\vec{\mu}_D)\right)}{3(2\pi)^{\frac{m}{2}} |\mathrm{A}_{\mathrm{D}}|^{\frac{1}{2}} p(x|\vec{\theta}, \vec{\pi}, J)}$$

$$f_L(x) = p(x \in L | x, \vec{\theta}, \vec{\pi}) = \frac{\pi_W p(x | x \in L, \vec{\theta}_L)}{p(x|\vec{\theta}, \vec{\pi}, J)} = \frac{\exp\left(-\frac{1}{2}(x-\vec{\mu}_L)^{\mathrm{T}} \mathrm{A}_{\mathrm{L}}^{-1}(x-\vec{\mu}_L)\right)}{3(2\pi)^{\frac{m}{2}} |\mathrm{A}_{\mathrm{L}}|^{\frac{1}{2}} p(x|\vec{\theta}, \vec{\pi}, J)}$$

dove m è il numero di fattori di conoscenza del giocatore artificiale, pari alla dimensione della variabile aleatoria x.
Definite

$$f_1(x) = \frac{f_W(x)}{f_L(x)}; \quad f_2(x) = \frac{f_D(x)}{f_L(x)}$$

la (3) si può scrivere come:

$$\frac{f_1(x) + \frac{1}{2} f_2(x)}{f_1(x) + f_2(x) + 1} \tag{6}$$

Questo è una valore compreso nell'intervallo [0,1] che indica il punteggio atteso da parte del giocatore che ha la mossa nella posizione x. Per rendere questo valore compatibile con il criterio di propagazione del

NegaMax e usare valori interi anziché reali, possiamo così definire la valutazione statica della posizione *x*:

$$BayesEval\,(x) = arr\left(512 \cdot \left[\frac{f_1(x) + \frac{1}{2} f_2(x)}{f_1(x) + f_2(x) + 1} - \frac{1}{2} \right]\right) \tag{7}$$

6.1.2 Apprendimento Bayesiano in GnuChess

Nel campo dei giochi finora l'apprendimento Bayesiano è stato sperimentato solo per il gioco dell'Othello, nel programma Bill (vedi paragrafo 3.3.1). Qui viene descritto un tentativo di applicazione al programma di Scacchi GnuChess.

Possiamo distinguere il processo di apprendimento in tre fasi:

1. generazione di un database di posizioni campione classificate nelle tre classi W, D e L;
2. analisi delle posizioni campione con le euristiche di valutazione proprie del programma GnuChess[2];
3. calcolo della distribuzione degli attributi nelle varie classi in base ai risultati della fase 2, da queste distribuzioni è definita la nuova funzione di valutazione statica prodotta dal processo di apprendimento.

Queste tre fasi vengono ripetute per tutti gli 11 stadi in cui GnuChess divide la partita in base alla variabile *stage* (vedi appendice A.5.1). In questo modo si ottengono 11 diverse funzioni di valutazione appositamente calibrate per le varie fasi della partita.

Per la fase 1 del processo di apprendimento sono state utilizzate le posizioni delle partite sulla difesa francese del database Deja Vu. Ogni posizione è stata assegnata ad una classe in base al risultato finale della partita in cui compare. La fase 2 è stata eseguita usando la funzione di valutazione statica di GnuChess che analizza ogni posizione secondo otto fattori di conoscenza: materiale, buona sistemazione dei

---

[2] Un tipo di analisi analoga su partite di maestri è descritta in [Har87a], [Har87b] e [Har89], con la differenza che Hartman si limita a calcolare le medie di ciascuna euristica di valutazione applicata alle posizioni del database per provare l'utilità dell'euristica, i dati così raccolti non vengono utilizzati per una modifica automatica della funzione di valutazione.

pezzi, spazio e mobilità, sicurezza del Re, controllo del centro, struttura pedonale, possibilità di attacco, relazione tra i pezzi (vedi paragrafo su GnuChess nel capitolo 2 e appendice A.5.3 per maggiori dettagli). In base ai risultati forniti dalla funzione di valutazione sono stati calcolati, come descritto dalle formule (4) e (5), i vettori delle medie e le matrici di covarianza che caratterizzano le funzioni di valutazione risultanti dal processo di apprendimento.

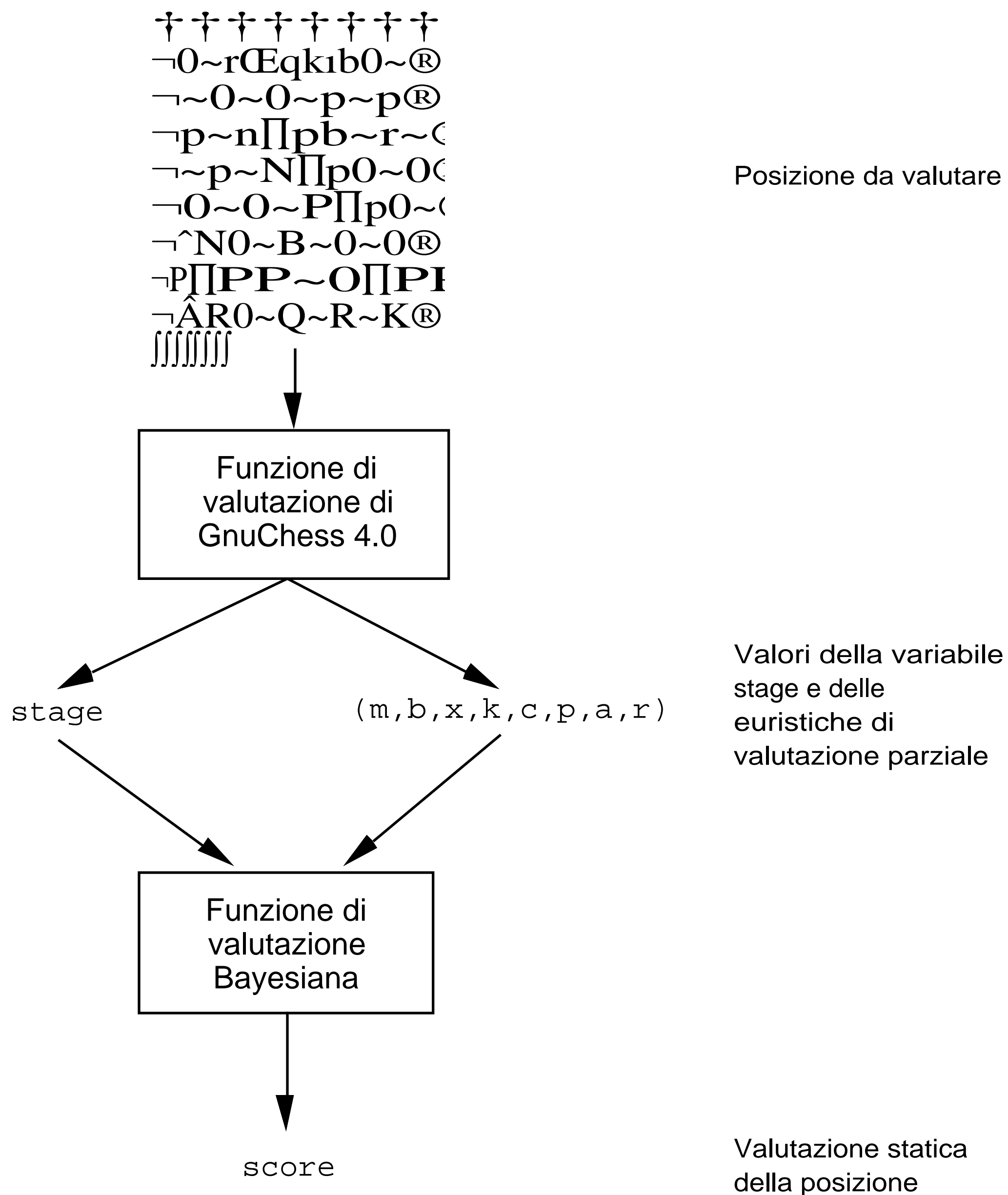


Figura 6.1. Schema del calcolo della nuova funzione di valutazione statica.

Nella figura 6.1 è illustrato schematicamente il calcolo dalla nuova funzione di valutazione statica durante una fase di gioco. Si può notare che la funzione di valutazione originaria di GnuChess viene ancora utilizzata per il calcolo delle euristiche di valutazione parziale e della variabile *stage*. In base a questi dati, la nuova componente della funzione di valutazione, corrispondente alla espressione (7), calcola poi lo *score* da attribuire alla posizione in esame.

Costo computazionale della nuova funzione di valutazione

Data la complessità dei calcoli, descritti dalla formula (6) e precedenti, che sono necessari a tempo di esecuzione per la funzione di valutazione creata con l'apprendimento Bayesiano, occorre limitare il più possibile questo overhead computazionale. Vediamo come viene calcolata la funzione di valutazione bayesiana descritta in (6), consideriamo prima la funzione $f_1(x)$:

$$\begin{aligned} f_1(x) = \frac{f_W(x)}{f_L(x)} &= \frac{|\mathrm{A_L}|^{\frac{1}{2}}}{|\mathrm{A_W}|^{\frac{1}{2}}} \cdot \frac{\exp\left(-\frac{1}{2}(x-\vec{\mu}_W)^{\mathrm{T}} \mathrm{A_W^{-1}} (x-\vec{\mu}_W)\right)}{\exp\left(-\frac{1}{2}(x-\vec{\mu}_L)^{\mathrm{T}} \mathrm{A_L^{-1}} (x-\vec{\mu}_L)\right)} = \\ &= k_{LW} \exp\left(\frac{(x-\vec{\mu}_L)^{\mathrm{T}} \mathrm{A_L^{-1}} (x-\vec{\mu}_L) - (x-\vec{\mu}_W)^{\mathrm{T}} \mathrm{A_W^{-1}} (x-\vec{\mu}_W)}{2}\right) \end{aligned} \tag{8}$$

dove $k_{LW} = \frac{|\mathrm{A_L}|^{\frac{1}{2}}}{|\mathrm{A_W}|^{\frac{1}{2}}}$ può essere precalcolata e considerata a tempo di esecuzione come una costante. Continuando lo svolgimento della parte rimanente della parte destra della (8) abbiamo:

$$\begin{aligned} &(x-\vec{\mu}_L)^{\mathrm{T}} \mathrm{A_L^{-1}} (x-\vec{\mu}_L) - (x-\vec{\mu}_W)^{\mathrm{T}} \mathrm{A_W^{-1}} (x-\vec{\mu}_W) = \\ &= x^{\mathrm{T}}(\mathrm{A_L^{-1}} - \mathrm{A_W^{-1}})x + \langle \vec{\mu}_W{}^{\mathrm{T}} \mathrm{A_W^{-1}} - \vec{\mu}_L{}^{\mathrm{T}} \mathrm{A_L^{-1}}, x \rangle + \langle x, \mathrm{A_W^{-1}}\vec{\mu}_W - \mathrm{A_L^{-1}}\vec{\mu}_L \rangle + \\ &+ \vec{\mu}_L{}^{\mathrm{T}} \mathrm{A_L^{-1}} \vec{\mu}_L - \vec{\mu}_W{}^{\mathrm{T}} \mathrm{A_W^{-1}} \vec{\mu}_W = x^{\mathrm{T}} \mathrm{D}_{LW} x + 2 \langle \vec{l}_{LW}, x \rangle + z_{LW} \end{aligned} \tag{9}$$

Nel calcolo sono state introdotte le variabili:

$$\mathrm{D}_{LW} = \mathrm{A_L^{-1}} - \mathrm{A_W^{-1}}, \qquad \vec{l}_{LW} = \mathrm{A_W^{-1}}\vec{\mu}_W - \mathrm{A_L^{-1}}\vec{\mu}_L, \qquad z_{LW} = \vec{\mu}_L{}^{\mathrm{T}} \mathrm{A_L^{-1}} \vec{\mu}_L - \vec{\mu}_W{}^{\mathrm{T}} \mathrm{A_W^{-1}} \vec{\mu}_W$$

Anche queste sono calcolate su costanti, e quindi a tempo di esecuzione questi valori si hanno a disposizione già precalcolati.
L'ultimo calcolo da svolgere nella (9) è, sfruttando la simmetria di D_{LW} :

$$x^T D_{LW} x = \langle D_{LW} x, x \rangle = \sum_{i=1}^{m} x_i \sum_{j=1}^{m} x_j d_{ij} = 2\sum_{i=1}^{m} x_i \sum_{j=1}^{i-1} x_j d_{ij} + \sum_{i=1}^{m} x_i^2 d_{ii}$$

Il costo computazionale complessivo per il calcolo di $f_1(x)$ a tempo di esecuzione, espresso calcolando il numero di operazioni matematiche necessarie, si può leggere nella tabella seguente.

Espressione	**Numero di operazioni**		
	add.	**molt.**	**exp.**
$x_i^2 \quad \forall i \in [1,m]$	0	m	0
$R_1 = \sum_{i=1}^{m} x_i^2 d_{ii}$	m-1	m	0
$R_2 = 2\sum_{i=1}^{m} x_i \sum_{j=1}^{i-1} x_j d_{ij}$	$\frac{m^2 - m}{2}$	$\frac{m^2 + m}{2}$	0
$R_3 = R_1 + R_2$	1	0	0
$R_4 = 2 \langle l_{LW}, x \rangle$	m-1	m	0
$k_{LW} \exp(R_3 + R_4 + z_{LW})$	2	1	1
Totale	$\frac{m^2}{2} + \frac{3m}{2} + 1$	$\frac{m^2}{2} + \frac{7m}{2} + 1$	1

I calcoli da svolgere per $f_2(x)$ sono del tutto analoghi. Si può però evitare di ricalcolare i quadrati degli attributi della posizione in input (prima riga della tabella 6.1), risparmiando m moltiplicazioni. Tenendo conto infine che per calcolare l'espressione (6) sono necessarie 3 addizioni e 1 moltiplicazione e che la funzione di valutazione originaria di GnuChess ha otto termini di conoscenza (quindi m=8), si ha che l'esecuzione della funzione di valutazione bayesiana comporta un costo aggiuntivo di:

$$m^2 + 3m + 5 = 89 \quad \text{addizioni}$$
$$m^2 + 6m + 3 = 115 \quad \text{moltiplicazioni}$$

2	esponenziali

Questo overhead, che non è certo trascurabile, si ha tutte le volte che viene chiamata la funzione di valutazione statica, a meno che non ci si trovi in un caso in cui si può usare un valore memorizzato nella tabella delle trasposizioni (vedi appendice A.3).

6.1.3 Sperimentazione

La fase di apprendimento è stata condotta utilizzando le 22.174 partite sulla apertura francese di Deja Vu, per un totale di 1.608.429 posizioni campione etichettate secondo il risultato delle partite in cui ciascuna posizione si è verificata.
Il giocatore artificiale ottenuto dalla fase di apprendimento (BayesPlayer) è stato confrontato con GnuChess tramite un match tematico sulla apertura francese. Il match ha mostrato la debolezza di BayesPlayer che ha totalizzato un solo punto su 20. Le cause di questa debolezza sono sostanzialmente due:

- calo di prestazioni provocato dalla nuova funzione di valutazione, che ha portato la velocità di analisi da 12.000÷13.000 nodi al secondo di GnuChess a circa 5.500 nodi al secondo;
- inaffidabilità della funzione di valutazione ottenuta dall'apprendimento Bayesiano: in diverse situazioni BayesPlayer gioca mosse incredibilmente deboli, spesso sacrificando materiale senza un motivo apparente. Un esempio di questo tipo di errore è illustrato nella figura 6.2.

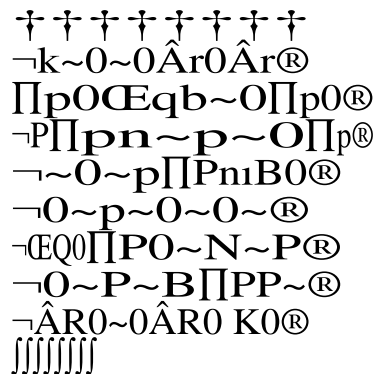


Figura 6.2. Una mossa debole di BayesPlayer. Nella posizione del diagramma gioca con il Bianco Bxc4?

Questa debolezza probabilmente è dovuta alla disomogeneità tra le posizioni analizzate dal programma mentre gioca e quelle nel campione di apprendimento. Ad esempio le posizioni provenienti del database in genere non presentano situazioni disperate per uno dei due colori (di solito un giocatore abbandona prima di prendere matto) e quando un giocatore è sotto di materiale spesso ha un sufficiente compenso posizionale. Le posizioni che invece il programma deve valutare possono presentare situazioni più strane e squilibrate, situazioni per le quali l'apprendimento dal database non dà una sufficiente preparazione.

Per cercare di porre rimedio ad entrambi questi problemi la funzione di valutazione *BayesEval*(x), descritta nel paragrafo precedente, è stata sostituita dalla nuova funzione:

$$MixedBayesEval(x) = \begin{cases} GnuChessEval(x) & \text{se } GnuChessEval(x) \notin [-100,100] \\ BayesEval(x) & \text{altrimenti} \end{cases}$$

Con questa nuova funzione di valutazione la costosa *BayesEval*(x) viene chiamata solo per valutare posizioni in cui GnuChess dà un vantaggio massimo di 100 punti ad uno dei due avversari. Per le altre posizioni viene mantenuta la valutazione di GnuChess, che fa semplicemente la somma delle otto valutazioni parziali. Il giocatore ottenuto con l'introduzione della funzione *MixedBayesEval*(x), chiamato MixedBayesPlayer, ha ottenuto 3,78 punti su 20 nel match contro GnuChess, un netto miglioramento rispetto a BayesPlayer, ma comunque molto lontano dalle prestazioni di GnuChess 4.0.

Nella tabella 6.2 sono riportati, oltre ai match già descritti, anche i risultati di altri esperimenti con versioni modificate di MixedBayesPlayer.

Match con apertura francese	**Risultato**	**Nodi/sec.**
BayesPlayer - GnuChess 4.0	1 - 19	5.500
MixedBayesPlayer - GnuChess 4.0	3,78 - 16,22	8.000
MixedBayesPlayerWD - GnuChess 4.0	1,06 - 18,94	8.000
MixedBayesPlayerGM - GnuChess 4.0	3,50 - 16,50	8.000
MixedBayesPlayerInd - GnuChess 4.0	0,10 - 19,90	9.000

Tabella 6.2 Esperimenti con apprendimento Bayesiano. Oltre al risultato di ciascun match è riportata indicativamente anche la velocità di analisi dei giocatori artificiali risultanti dall'apprendimento.

MixedBayesPlayerWD è stato ottenuto non considerando, durante l'apprendimento, le posizioni delle partite terminate in parità, per cui le classi in cui sono divise le posizioni campione sono solo W e L.
MixedBayesPlayerGM è stato ottenuto considerando, durante l'apprendimento, solo le posizioni delle partite giocate da grandi maestri.
Infine MixedBayesPlayerInd è stato ottenuto considerando tra loro indipendenti i fattori di conoscenza, per cui la distribuzione dei valori dei termini di conoscenza all'interno di ciascuna classe non è più approssimata con una legge normale multivariata ad otto dimensioni, ma semplicemente con il prodotto di otto distribuzioni normali.
I risultati ottenuti da questi tre ultimi giocatori artificiali ci mostrano tre cose:

- come era logico aspettarsi, le patte non possono essere trascurate nella fase di apprendimento per il gioco degli Scacchi;
- la netta sconfitta di MixedBayesPlayer non è dovuta alla qualità delle partite da cui sono state tratte le posizioni campione, infatti MixedBayesPlayer ha ottenuto un risultato addirittura inferiore, ma sostanzialmente analogo, a quello ottenuto da MixedBayes-Player;
- tenere conto della non linearità delle relazioni tra i vari fattori di conoscenza può essere il vero potenziale vantaggio della funzione di valutazione bayesiana. Infatti MixedBayesPlayerInd, pur essendo più veloce di MixedBayesPlayer, ha ottenuto un risultato molto più modesto contro GnuChess, appena 0,10 punti su 20.

In conclusione si può dire che l'apprendimento Bayesiano non ha dato risultati positivi nell'apprendimento da database di partite di Scacchi.

La causa di questo insuccesso va probabilmente ricercata nel tipo di informazioni fornite all'algoritmo di apprendimento. Usare partite di alto livello non ha infatti portato a risultati soddisfacenti. L'alternativa potrebbe essere l'uso di posizioni ricavate da partite di basso livello o addirittura ottenute con gioco casuale, in modo da documentare un più vasto spettro di situazioni che si possono verificare durante l'analisi svolta da un giocatore artificiale. Non sembra invece un handicap eccessivo la perdita di prestazioni dovuta alla maggiore complessità della funzione di valutazione: al massimo si perde circa il 55% di velocità di analisi, perdita che può essere compensata da una funzione più precisa, che tenga conto della non linearità delle relazioni che ci sono tra le varie euristiche di valutazione.

6.2 Algoritmi genetici

6.2.1 Algoritmi genetici nei giochi a informazione completa

Gli algoritmi genetici, descritti brevemente nel paragrafo 3.4, costituiscono un diverso approccio al problema dell'apprendimento da database per lo sviluppo della funzione di valutazione, rispetto all'apprendimento Bayesiano.
Innanzi tutto la struttura della funzione di valutazione che cercheremo di ottimizzare con questo tipo di algoritmi è quella classica, descritta nel paragrafo 2.4, costituita dalla somma pesata di particolari euristiche parziali di valutazione:

$$f(p) = \sum_{i=1}^{N} weight_i \cdot A_i(p)$$

Il compito dell'algoritmo genetico è quello di stabilire la combinazione di pesi *weight* per cui la funzione di valutazione è migliore.
Il processo di ottimizzazione introdotto dagli algoritmi genetici prevede:

- una popolazione di individui, nel nostro caso di giocatori artificiali caratterizzati dal vettore dei pesi *weight*;
- la codifica dei pesi *weight* in stringhe di bit, che costituiscono il codice genetico di ciascun individuo;
- la valutazione di ciascun individuo della popolazione tramite una apposita funzione (*fitness function*) che misura l'adeguatezza di

ogni individuo al compito che deve svolgere. Per motivi di efficienza la *fitness function* è costituita di solito da un test su posizioni campione, piuttosto che da un torneo tra gli individui della stessa popolazione;

- accoppiamento degli individui in modo che la frequenza di accoppiamento di ciascun individuo sia proporzionale al valore di *fitness*;
- generazione di due individui figli per ogni coppia determinata nella fase precedente. Il codice genetico dei figli viene determinato tramite il cross-over dei codici dei genitori e, per aumentare la varietà complessiva dei codici genetici, tramite la sporadica mutazione casuale di alcuni bit (fig. 6.3).

  Stabilite con PC e PM rispettivamente le probabilità di cross-over e di mutazione il codice genetico di ciascun figlio *x*, generato da *g1* e *g2* può essere così definito:

```
for(i=0; i<codeLenght; i++)
{
   if (randValue1<PC)
         // se cross-over...
         if (i>l1 && i<l2)
               // se i è in un certo intervallo, definito
               // casualmente, prende il codice da g1
               code[x][i]=code[g1][i];
         else
               // altrimenti prende il codice da g2
               code[x][i]=code[g1][i];
   else
         // se non cross-over
         code[x][i]=code[g1][i];
   if (randValue2<PM)
         // mutazione
         code[x][i]=~code[x][i];
}
```

```
g1  1 0 0 0 0 1 0 1 1 1 1 0 1 0 1 0 0 0 0 ... 0 1 1 0 0
    0       11                 12                 N-1
g2  1 1 1 1 0 0 1 0 0 0 1 1 0 0 1 0 0 1 1 ... 1 0 1 0 0

                       cross-over

x   1 1 1 1 0 1 0 1 1 1 1 0 1 0 1 0 0 1 1 ... 1 0 1 0 0

                       mutation

x   1 1 1 0 0 1 0 1 1 1 1 0 1 0 1 0 0 1 1 ... 1 0 1 1 0
```

Figura 6.3. Esempio di riproduzione.

Le fasi di valutazione degli individui e creazione di una nuova generazione vengono ripetute finché non si ritiene di aver individuato una funzione di valutazione sufficientemente buona, o comunque quando si osserva che l'algoritmo si è stabilizzato.

6.2.2 *Algoritmi genetici applicati a GnuChess*

Nel caso di GnuChess i pesi da ottimizzare sono quelli relativi agli otto fattori di conoscenza utilizzati anche per l'apprendimento Bayesiano (vedi paragrafo 2.5.2 e appendice A.5.3).
Ogni popolazione è costituita da 30 funzioni di valutazione. Ad ogni individuo è associata una stringa di 64 bit che codifica il valore dei pesi dei fattori di conoscenza. Ad ogni fattore di conoscenza sono assegnati otto bit che codificano un valore intero compreso tra 0 e 255.
La *fitness function* è costituita da un test su insiemi di posizioni, sfruttando la caratteristica di GnuChess di eseguire automaticamente test su posizioni in notazione Forsyth (vedi appendice A.6). Le posizioni e il loro numero saranno scelti di volta in volta in base al tipo di esperimento che sarà condotto. Come già discusso nel capitolo 3, la *fitness function* costituisce il collo di bottiglia degli algoritmi genetici applicati ai giochi. Perciò per ogni posizione campione, ogni individuo tenterà di trovare la mossa migliore con una ricerca di una sola

semimossa di profondità più la profondità raggiunta dalla ricerca quiescente. Questa è la ricerca più rapida, ma comunque significativa, che si possa eseguire. Tuttavia la valutazione di una popolazione di 30 individui con un campione di 500 posizioni richiede comunque circa 2 ore di elaborazione. Per amplificare il divario tra individui con prestazioni diverse la *fitness function* è così definita:

$$FitnessFunction = \begin{cases} 1 & \text{se } npos(I) < \dfrac{N}{5} \\ \dfrac{\left(npos(I) - \dfrac{N}{5}\right)}{N} + 1 & \text{altrimenti} \end{cases}$$

dove $npos(I)$ è il numero di mosse corrette scelte dall'individuo I e N il numero totale di posizioni campione.
Per la riproduzione sono stati impostati i valori PC=0,6 e PM=0,05.

6.2.3 Sperimentazione

Un primo esperimento è stato condotto scegliendo come posizioni campione un insieme di 500 posizioni delle partite del database Deja Vu riguardanti la Difesa Francese. Queste posizioni sono state scelte casualmente tra le posizioni comprese tra la 10ª e la 50ª mossa di ciascuna partita, in modo da escludere le prime mosse di apertura per concentrare l'apprendimento sulle fasi di mediogioco e finale. Le mosse giocate nelle partite del database nelle posizioni scelte sono state considerate le mosse corrette per queste posizioni. Per evitare di assumere come corrette delle mosse deboli, sono state scelte solo posizioni di partite tra grandi maestri, e nelle posizioni scelte la mossa sta sempre al giocatore che poi ha vinto la partita. Ciò non toglie, comunque, che in qualche caso nelle posizioni campione ci siano mosse migliori di quelle che compaiono nel database. Il giocatore artificiale risultato dall'apprendimento è stato chiamato Genetic 1.
Un secondo esperimento è stato condotto concentrando maggiormente l'attenzione dell'apprendimento nella fase di mediogioco immediatamente successiva all'apertura. Per questo scopo le posizioni campione sono state scelte tra quelle la cui variabile *stage* di GnuChess (vedi appendice A.5.1) è compresa tra i valori 2 e 4. Dato che le posizioni che

soddisfano sono più simili tra loro di quanto lo sono quelle scelte tra la 10ª e la 50ª mossa, l'insieme di posizioni su cui viene calcolata la *fitness function* è stato ridotto da 500 a 100 unità. Inoltre, dato che l'apprendimento avviene solo su posizioni con 2 ≤ *stage* ≤ 4, in fase di gioco il giocatore artificiale risultante dall'apprendimento (che chiameremo Genetic 2) userà i pesi ricavati dall'algoritmo genetico solo per valutare posizioni in cui 2 ≤ *stage* ≤ 4, mentre per le altre posizioni userà la normale funzione di valutazione di GnuChess.
Nella tabella 6.3 sono riportati i risultati dei match, nei quali entrambi gli avversari utilizzano il libro di aperture standard di GnuChess, limitato alla sola Difesa Francese.

Risultati con apertura francese	
Genetic 1 - GnuChess 4.0 con libro	9,50 - 10,50
Genetic 2 - GnuChess 4.0 con libro	10,93 - 9,07

Tabella 6.3. Esperimenti con algoritmo genetico.

Genetic 1 ha perso di misura contro GnuChess, mentre Genetic 2 ha vinto totalizzando quasi 11 punti su 20. Evidentemente per l'algoritmo genetico è più difficile ottimizzare i pesi dei fattori di conoscenza analizzando un insieme eterogeneo di posizioni, come quello utilizzato da Genetic 1. Tuttavia il risultato di Genetic 1 non è da disprezzare dato che per trovare i pesi usati da questo giocatore artificiale sono state necessarie circa 20 iterazioni dell'algoritmo genetico per un totale di circa 40 ore di elaborazione, mentre ai programmatori di GnuChess sarà servito certamente più tempo.
Restringendo il campo di applicazione dell'algoritmo genetico alle posizioni più vicine all'apertura, Genetic 2 ha ottenuto risultati migliori, pur usando per l'apprendimento un insieme di posizioni più piccolo. L'aver trovato una combinazione di pesi migliore, per le posizioni della Difesa Francese con 2 ≤ *stage* ≤ 4, è stato sufficiente a Genetic 2 per battere GnuChess, anche se non con un punteggio schiacciante.
Per vedere quanto i pesi determinati con l'algoritmo genetico siano dipendenti dall'apertura da cui sono state tratte le posizioni campione, sono stati effettuati due test tematici sulla apertura Spagnola (codici ECO C60-C99) e sul Gambetto di Re (C30-C39) tra Genetic 2 e GnuChess

4.0 (tabella 6.4). Genetic 2 utilizza i pesi trovati per le posizioni tratte dalla Difesa Francese.

Risultati con apertura spagnola	
Genetic 2 - GnuChess 4.0 con libro	7,74 - 12,26
Risultati con gambetto di Re	
Genetic 2 - GnuChess 4.0 con libro	12,43 - 7,57

Tabella 6.4. Match di Genetic 2 su altre aperture.

I risultati ottenuti sono molto contrastanti, Genetic 2 vince nettamente con il Gambetto di Re e perde, altrettanto nettamente, con la Spagnola. Da questi risultati si può dedurre che esiste una relazione tra l'apertura da cui sono state tratte le posizioni usate dalla Fitness Function e l'attitudine del programma a giocare in certe aperture meglio che in altre. Evidentemente, però, il fatto di utilizzare posizioni con $2 \leq stage \leq 4$ come posizioni di test (quindi più di mediogioco che di apertura) e di bilanciare i pesi di euristiche abbastanza generali (come materiale, struttura pedonale, ecc.) consente che i pesi calibrati per un certo tipo di apertura siano adatti anche per altre. Nel nostro caso si è verificato addirittura che i pesi calibrati per la Difesa Francese abbiano dato il risultato migliore con il Gambetto di Re, un tipo di apertura assai diversa. Forse le cose sarebbero andate diversamente se fossero stati calibrati pesi di euristiche più specifiche, cioè che avessero tenuto conto di caratteristiche più dipendenti dall'apertura, ad esempio: quanto vale l'Alfiere campochiaro rispetto a quello camposcuro, è meglio doppiare i Pedoni su una colonna piuttosto che su un'altra, ecc.

Capitolo 7

Conclusioni

In questa tesi si è visto come è possibile ricavare da database di partite conoscenza utile per un giocatore artificiale di Scacchi. In particolare è stato preso in considerazione un giocatore artificiale che adotta una strategia definita da Shannon di tipo A (forza bruta) con un algoritmo di ricerca di tipo AlphaBeta.

7.1 Lavoro svolto

Sono stati individuati tre diversi modi di utilizzare le informazioni del database di partite:

- creare automaticamente un libro di aperture;
- preordinare le mosse durante la visita dell'albero di gioco;
- sviluppare la funzione di valutazione statica con tecniche di Machine Learning basate sui dati del database.

I test condotti hanno mostrato la validità del libro di aperture creato in maniera automatica dalle partite del database: la versione di GnuChess fornita del nuovo libro si è infatti rivelata più forte di GnuChess senza libro di aperture, e all'incirca dello stesso livello di GnuChess con il suo libro di aperture standard.

Un miglioramento si è ottenuto con l'introduzione di due diverse tecniche di preordinamento delle mosse che permettono di incrementare significativamente le prestazioni dell'algoritmo AlphaBeta. Anche queste due tecniche si basano sull'uso di informazioni del database per sintetizzare conoscenza specifica del gioco degli Scacchi. I test su un insieme di posizioni riguardanti l'apertura francese hanno

mostrato che nelle situazioni nelle quali queste tecniche sono applicabili i tagli effettuati dall'algoritmo di ricerca di GnuChess arrivano fino al 30% dei nodi visitati da GnuChess originale. Questo aumento di prestazioni ha poi mostrato i suoi effetti positivi nei match tra la versione di GnuChess modificata con le tecniche di preordinamento e GnuChess originale. Inoltre è stato rilevato un ulteriore miglioramento attraverso la combinazione delle informazioni del database con quelle ricavate dall'esperienza del programma di gioco accumulata in partite di allenamento.

Infine sono state sperimentate due tecniche di Machine Learning per lo sviluppo della funzione di valutazione statica tramite l'osservazione dei dati contenuti nel database di partite: l'apprendimento Bayesiano e gli algoritmi genetici.

Gli esperimenti condotti sull'apprendimento Bayesiano hanno dato esito negativo, a dispetto degli ottimi risultati ottenuti nel gioco dell'Othello [LeeMah90]. I motivi di questo insuccesso dovrebbero essere principalmente due: i database di partite non sono la migliore fonte di conoscenza per questo tipo di apprendimento, e l'insieme di euristiche di valutazione usate da GnuChess non si presta all'analisi statistica che viene condotta nell'apprendimento Bayesiano. Comunque dai test viene confermato che considerare la non linearità delle relazioni tra i termini di conoscenza dà dei potenziali vantaggi rispetto alla loro semplice combinazione lineare. Infatti, nei test, il giocatore Bayesiano che non fa uso della matrice di covarianza dei termini di conoscenza ha dato risultati molto più negativi del giocatore Bayesiano che ne tiene conto.

Gli algoritmi genetici, invece, hanno confermato di essere un buon metodo di ottimizzazione per le funzioni di valutazione statica di giochi ad informazione completa. Infatti, il giocatore artificiale risultante dall'algoritmo genetico ha mostrato nei test un leggero miglioramento rispetto alla versione originale di GnuChess 4.0. Molto probabilmente questo miglioramento non è potuto essere più forte, non per inadeguatezza dell'algoritmo genetico utilizzato, ma per la funzione di valutazione di GnuChess che combina le varie euristiche di valutazione in maniera già ben equilibrata, e quindi difficilmente migliorabile.

Per valutare il miglioramento complessivo che si ha mettendo insieme tutte le tecniche sviluppate in questa tesi (escluso l'apprendimento Bayesiano), è stato giocato un match (tabella 7.1) tra GnuChess 4.0 originale, con il proprio libro di aperture standard, e la versione di

GnuChess con il libro di apertura creato dal database ed esteso tramite feedback, le tecniche di preordinamento delle mosse introdotte nel capitolo 5, i pesi delle euristiche di valutazione parziali bilanciati con l'algoritmo genetico descritto nel capitolo 6 per il giocatore artificiale Genetic 2.

Apertura francese	
OPHISTA +FB+Genetic 2 - GnuChess 4.0 con libro	12,78 - 7,22

Tabella 7.1. Esperimento conclusivo.

Da questo match si può stimare che la forza di gioco di GnuChess è aumentata di circa 100 punti ELO (vedi [Elo86], pag. 19). Inoltre si osserva che il risultato ottenuto sommando le varie euristiche è stato superiore a tutti i risultati ottenuti con l'impiego di solo alcune di esse (vedi tabelle 4.5, 5.1, 5.4, 5.7 e 6.3).

7.2 Prospettive future

Questa tesi ha dimostrato come è possibile migliorare un giocatore artificiale di Scacchi con l'uso appropriato delle informazioni contenute in database di partite, tuttavia le metodologie usate si prestano ad un gran numero di varianti e sviluppi che potrebbero essere presi in considerazione in lavori futuri.
Ad esempio, il database di partite potrebbe essere più esteso di quello utilizzato per questa tesi. Si potrebbe anche pensare di ricavare un grande database costantemente aggiornato scaricando le partite dai vari nodi scacchistici di Internet, facendo attenzione alle inevitabilmente numerose partite doppie. Se le partite fossero commentate si potrebbero usare per l'apprendimento non solo le mosse realmente giocate in partita, ma anche le analisi e le valutazioni dell'autore dei commenti; con questo metodo un giocatore artificiale potrebbe apprendere non solo aperture già sperimentate, ma anche nuove idee non ancora provate in partite di alto livello.
Anche il giocatore artificiale sul quale sperimentare le tecniche di apprendimento potrebbe essere diverso da GnuChess. Ad esempio, potrebbe essere interessante provare l'efficacia della tecnica delle

HistoryTable (illustrata nel paragrafo 5.2) inserita in un giocatore artificiale parallelo a ricerca distribuita. È noto infatti che molte euristiche di ordinamento delle mosse, tra cui le mosse killer e le tabelle delle trasposizioni, hanno difficoltà ad essere applicate nei giocatori artificiali paralleli con gli stessi vantaggi che si hanno nei giocatori sequenziali, a causa della mancanza di una visione globale dell'analisi dell'albero di gioco. Le *HistoryTable*, al contrario delle euristiche sopra citate, possono essere facilmente sfruttate localmente, senza conoscere lo stato complessivo dell'analisi in corso.
Per quanto riguarda lo sviluppo della funzione di valutazione con tecniche di ML (trattato nel capitolo 6), si potrebbe tentare di migliorare l'efficacia dell'apprendimento Bayesiano principalmente in due modi: usare posizioni campione diverse e modificare la struttura della funzione di valutazione del giocatore artificiale che apprende. Le posizioni campione potrebbero essere generate in maniera simile a quanto descritto in [LeeMah90] (vedi paragrafo 6.1.1), cioè con una generazione quasi casuale, piuttosto che tratte da database di partite. Così si potrebbe fornire un migliore allenamento per le posizioni che il giocatore artificiale si troverà a valutare durante l'analisi, che sono spesso molto diverse da quelle che si possono trovare nei database. Infine, la riformulazione della funzione di valutazione con euristiche facilmente approssimabili con funzioni normali, che sono usate dall'apprendimento Bayesiano, potrebbe garantire un miglior grado di precisione e quindi una maggiore accuratezza nel calcolo della funzione di valutazione.

Appendice A

GnuChess 4.0

In questa appendice sono illustrate in maniera dettagliata le caratteristiche principali della struttura del programma Gnuchess 4.0.

A.1 Rappresentazione dello stato del gioco e scelta della mossa

Per rappresentare completamente una posizione (nel senso di dare tutte le informazioni necessarie per il prosieguo della partita) sono necessari i seguenti dati: case occupate da ciascun pezzo, giocatore che ha la mossa, possibilità di arrocco di ciascun giocatore, la casa in cui è possibile una presa *en passant*, sono anche necessarie informazioni riguardanti le mosse precedenti per poter riconoscere i casi di patta per triplice ripetizione o per la regola delle 50 mosse ed eventualmente le informazioni riguardanti il controllo di tempo.
In GnuChess i tipi di pezzo sono rappresentati con le costanti *pawn*, *knight*, *bishop*, *rook*, *queen*, *king* e i colori con *white*, *black* e *neutral* (per le case vuote). La posizione dei pezzi viene mantenuta in due diverse strutture che vengono usate alternativamente in vari punti del programma per motivi di efficienza: *PieceList* [2] che per ciascun giocatore contiene la lista di case che i suoi pezzi occupano, e la coppia di array *board*[64] e *color*[64] che contengono rispettivamente il tipo e il colore dei pezzi presenti sulla scacchiera. Le case sono codificate in modo che la “a1” corrisponda allo 0, le altre sono numerate da sinistra a destra e dal basso in alto fino alla casa “h8”, che viene codificata con il numero 63. Le variabili *castld*[2] e *Mvboard*[2][64] indicano rispettivamente se un giocatore ha già arroccato e per ogni casa quante volte

il pezzo che vi trova è stato mosso, entrambe contribuiscono a stabilire se un giocatore può ancora arroccare o no. La variabile *epsquare* indica la casa in cui è possibile la presa *en passant*.
La storia della partita viene memorizzata nell'array GameList con elementi del tipo:

```
struct GameRec
{
     unsigned short    gmove;  // codifica della mossa
     short             score;  // valutazione di GnuChess
     long              time;   // tempo impiegato per
                               // giocare la mossa
     short             piece;  // tipo di pezzo catturato
     short             color;  // colore del pezzo catturato
     short             flags;  // informazioni supplementari
     short             Game50; // numero dell'ultima mossa
                               //  di cattura o di pedone
     long              nodes;  // numero di nodi visitati
     unsigned long     hashkey;// chiave per la tabella
                               // delle trasposizioni
     unsigned long     hashbd; // codice di controllo per la
                               //tabella delle trasposizioni
     short             epssq;  // casa di presa en passant
                               // dopo l'esecuzione della
                               // mossa, -1 se non possibile
}
```

Il controllo del tempo in GnuChess viene effettuato mediante la struttura *TimeControl*, del tipo:

```
struct TimeControlRec
{
     short moves[2]; // mosse da giocare da ciascun colore
     long  clock[2]; // tempo a disposizione in secondi
}
struct TimeControlRec TimeControl;
/* TimeControl.moves[player], TimeControl.clock[player] */
```

Prima di iniziare la ricerca il programma calcola il tempo da dedicarvi, in base al tempo a disposizione e al numero di mosse che gli restano da giocare per superare il controllo di tempo, e la memorizza nella variabile *ResponseTime* ed inoltre calcola un tempo straordinario (in *ExtraTime*) da utilizzare nel caso ci sia bisogno (per es. in caso di fallimento della prima ricerca dell'Aspiration Search). Quando viene superato il tempo assegnato alla mossa (*ResponseTime* + *ExtraTime*) GnuChess interrompe la ricerca in qualsiasi stato essa si trovi e gioca la mossa che fino a quel momento sembra essere la migliore.
Nella figura A.1 è riportato lo schema della funzione *SelectMove*( ), che ha il compito di scegliere la mossa da giocare, scegliendo tra le mosse del libro di apertura, se ce ne sono, oppure utilizzando l'algoritmo di ricerca. Nella figura A.2, invece, è illustrato il ciclo di base di GnuChess.

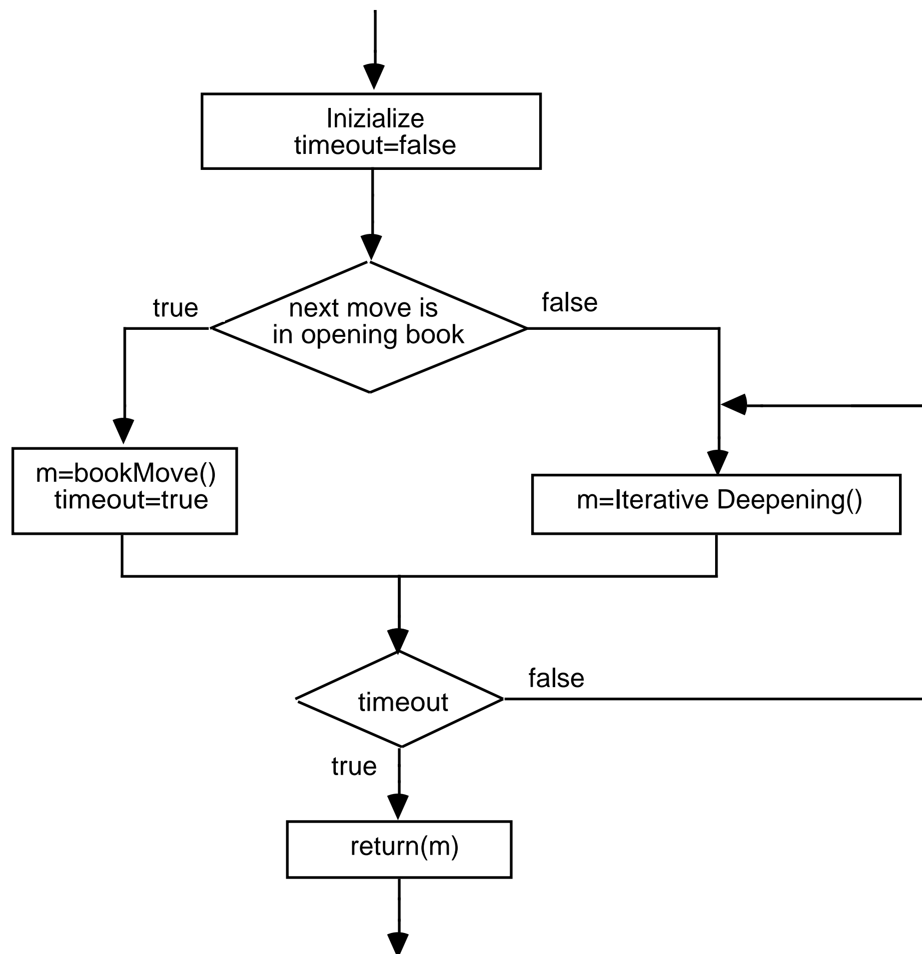


Figura A.1. Schema della funzione SelectMove( ).

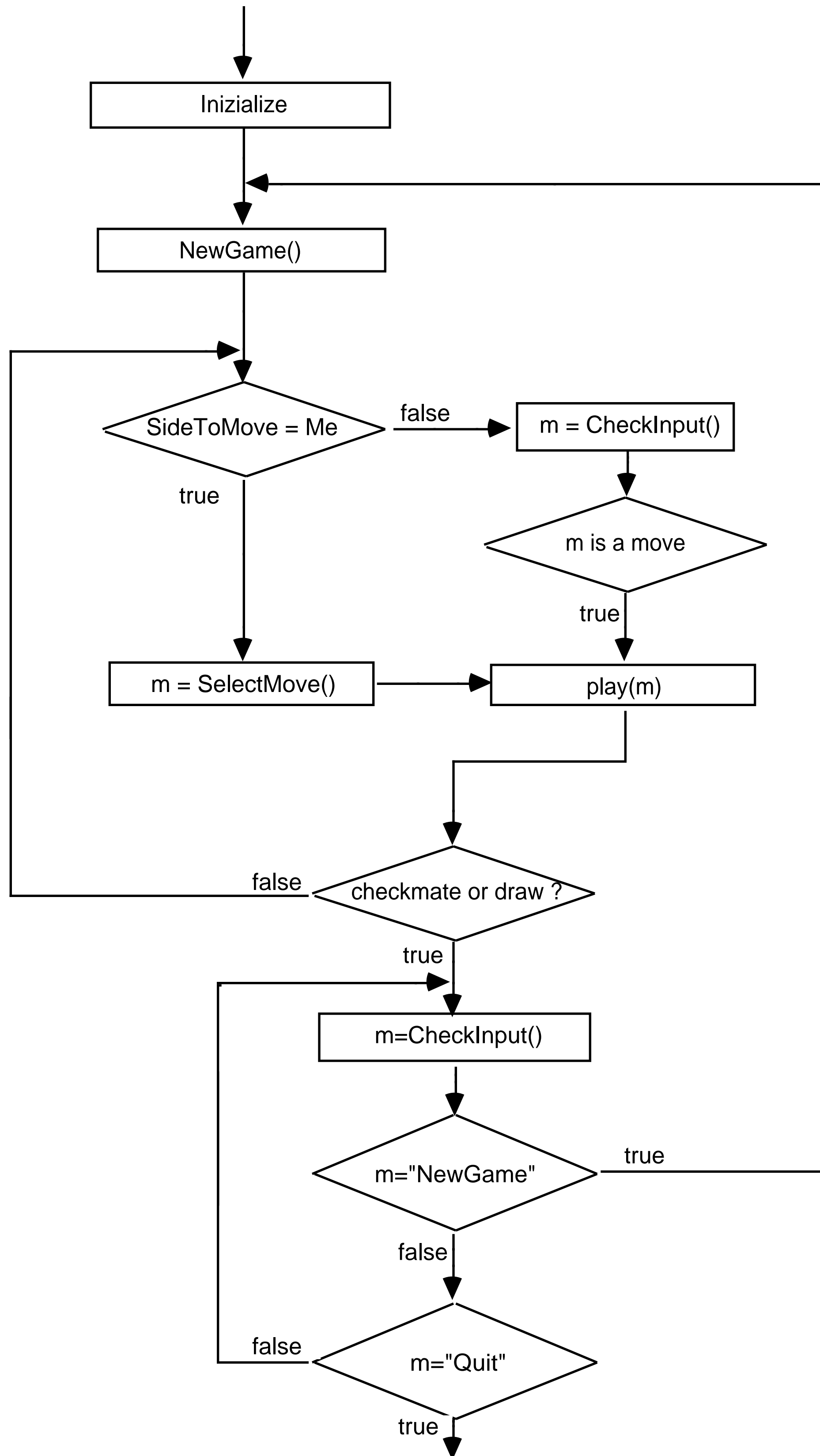


Figura A.2. Ciclo di base di GnuChess 4.0.

A.2 Generatore di mosse

Il generatore di mosse di GnuChess è una derivazione di un algoritmo pensato per un generatore hardware [San89]. L'idea di base di questo algoritmo è di precalcolare quanti più dati possibile, in modo da guadagnare in efficienza a tempo di esecuzione. Vengono precalcolate tutte le mosse possibili, di ciascun tipo di pezzo da qualsiasi casa di partenza, in un array del tipo:

```
struct sqdata
{
     short nextpos;
     short nextdir;
};
struct sqdata posdata[8][64][64];
/* posdata[piecetype][fromsquare][destinationsquare] */
```

`piecetype` può avere il significato di Pedone Bianco, Pedone Nero, Cavallo, Alfiere, Torre, Donna o Re, la distinzione tra Pedone Bianco e Nero è necessaria dato che questi hanno movimenti completamente diversi tra loro, dovendo andare in direzioni opposte. L'array viene utilizzato nel seguente modo: la prima mossa per il tipo di pezzo `piecetype` situato in `fromsquare` è memorizzata in

```
posdata[piecetype][fromsquare][fromsquare].nextpos
```

chiamiamo `f` e `t` le case di partenza e arrivo di questa mossa. Se `t` è libera possiamo proseguire nella stessa direzione, e la prossima mossa si troverà in

```
posdata[piecetype][fromsquare][t].nextpos
```

se invece la casa `t` è occupata la prossima mossa sarà in

```
posdata[piecetype][fromsquare][t].nextdir
```

Il procedimento si ripete fino a quando non si ha che `t = fromsquare`: questa è ovviamente una mossa illegale, e serve a indicare che tutte le mosse per il pezzo in `fromsquare` sono state considerate.

Condizione	Bonus
Variante principale	2000
Cattura dell'ultimo pezzo mosso	500
Cattura di una Donna	1100
Cattura di una Torre	550
Cattura di un Alfiere	355
Cattura di un Cavallo	350
Cattura di un Pedone	100
Promozione a Donna	800
Promozione a Cavallo	600
Promozione a Torre	550
Promozione a Alfiere	500
Spinta di Pedone in settima traversa	600
Spinta di Pedone in sesta traversa	400

Tabella A.1. Bonus per il pre-ordinamento delle mosse.

Il calcolo dell'array `posdata` viene effettuato dalla procedura Inizialize_moves( ) all'avvio di GnuChess 4.0, e viene letto, durante l'analisi dell'albero di gioco, dalle procedure MoveList(ply) e CaptureList(ply), la prima inserisce nell'albero di gioco tutte le mosse pseudo-legali della posizione correntemente analizzata alla profondità ply, mentre CaptureList(ply), che serve per la ricerca quiescente, considera solo le mosse di cattura, scacco e promozione.

Contemporaneamente all'inserimento delle mosse nell'albero di gioco, viene assegnato a ciascuna mossa un bonus che sarà utile per ordinare le mosse stesse, rendendo così più efficiente l'algoritmo AlphaBeta. Le euristiche impiegate per assegnare questi bonus sono studiate per ordinare le mosse secondo i seguenti criteri:

1. Mosse della variante principale (cioè le mosse risultate migliori nell'iterazione precedente dell'algoritmo alfabeta)
2. Mosse di cattura dell'ultimo pezzo mosso
3. Mosse di cattura ordinate secondo il peso del pezzo catturato
4. Mosse di promozione e spinte di Pedoni vicini alla promozione

Nella tabella A.1 sono riportati i bonus assegnati alle mosse secondo le condizioni che soddisfano. Se una mossa soddisfa più condizioni, i bonus vengono sommati.

A.3 Tabella delle trasposizioni

La tabella delle trasposizioni è formata da due tabelle hash, una per le posizioni in cui il Bianco ha la mossa e una per le posizioni in cui la mossa è al Nero. Queste due tabelle hash vengono dimensionate, in fase di inizializzazione del programma, in base alla memoria disponibile. La funzione hash che associa alle posizioni una entry della tabella è quella proposta in [Zob70]. In fase di inizializzazione vengono calcolati gli elementi dell'array

```
struct {unsigned long key, bd} haschcode[2][6][64]
```

che dato un colore, un tipo di pezzo, una casa, restituisce una coppia di numeri pseudo casuali: key che serve per calcolare la entry associata alla posizione, e bd che serve per calcolare il codice di controllo.
Quindi per ogni posizione che sarà incontrata nella ricerca possono essere calcolati i valori hashkey (entry nella tabella) e hashbd (codice di controllo) tramite un'operazione di or esclusivo dei valori pseudo casuali associati a ciascun pezzo sulla scacchiera:

```
void h()
{
      hashkey=0;
      hashbd=0;
      for(i=0;i<64;i++)
            if (color[sq]!=neutral)
            {
                  hashkey^= hashcode[color[sq]][board[sq]][sq].key;
                  hashbd ^= hashcode[color[sq]][board[sq]][sq].bd;
            }
}
```

In realtà la funzione h( ) è necessaria solo all'inizio della partita per la posizione iniziale. I valori hash delle altre posizioni possono essere infatti calcolati incrementalmente in base ai valori della posizione precedente nell'albero di gioco. Infatti, grazie alle proprietà dell'operatore xor, prima di eseguire una mossa dalla casa `f` alla casa `t`, basta

eseguire le seguenti operazioni per avere il valore hash della posizione successiva:

```
// togli il pezzo mosso dalla casa di partenza
hashkey^= hashcode[color[f]][board[f]][f].key;
// aggiungi il pezzo mosso nella casa di arrivo
if (promote)
      hashkey^= hashcode[color[f]][prom_piece][t].key;
else
      hashkey^= hashcode[color[f]][board[f]][t].key;
if (capture)
// togli il pezzo catturato nella casa di arrivo
      hashkey^= hashcode[color[t]][board[t]][t].key;
```

Le operazioni per il valore hashbd sono del tutto analoghe.
Ciascuna entry della tabella delle trasposizioni è del tipo:

```
struct hashentry
{
     unsigned long hashbd; // codice di controllo
     unsigned char flags;  // informazioni aggiuntive
     char depth;           // profondità analisi
     unsigned short score; // score della posizione
     unsigned short mv;    // mossa da giocare
     unsigned short age;   // anzianità della posizione
};
```

Il campo `flags` contiene dei flag che indicano se lo score è il reale valore negamax della posizione oppure è un limite superiore o inferiore, se la mossa suggerita nel campo `mv` appartiene al libro delle aperture, se nella posizione relativa alla entry è possibile giocare una mossa di arrocco. Il campo `age` è utile per gestire il rimpiazzamento delle entry più vecchie: quando una nuova posizione non trova la propria entry libera e neppure una delle sette entry successive, allora la più vecchia di queste otto entry viene rimpiazzata.
In fase di visita dell'albero di gioco, visitando un nodo GnuChess cerca la posizione corrispondente nella tabella delle trasposizioni. Se la posizione è presente lo score associato nella tabella viene usato se sono verificate le seguenti condizioni:

- la profondità con cui è stato calcolato lo score è superiore o uguale a quella richiesta dalla ricerca in corso, e
- lo score è esatto oppure la limitazione superiore o inferiore che fornisce è sufficiente per rendere inutile la ricerca nel sottoalbero che ha per radice il nodo che si sta visitando. Precisamente quando `score` è un limite inferiore e `score` $> \beta$, oppure quando `score` è una limitazione superiore e `score` $< \alpha$.

A.4 Libro delle aperture

GnuChess 4.0 è in grado di usare due tipi di file come libro di aperture: un file con informazioni codificate (file binario), e uno in formato testuale (file ASCII). Il secondo tipo di file è quello che è stato usato per questa tesi. In esso sono memorizzate in notazione PGN (Portable Game Notation) varie linee di gioco alternate a linee di commento (vedi fig. A.3) utili per eventuali modifiche al file che possono essere eseguite con un qualsiasi editore di testo.

```
[ECO B57 Sicilian: Sozin, Benkö variation]
e4 c5  Nf3 d6  d4 cd  Nd4 Nf6  Nc3 Nc6  Bc4 Qb6
[ECO B57 Sicilian: Magnus Smith trap]
e4 c5  Nf3 d6  d4 cd  Nd4 Nf6  Nc3 Nc6  Bc4 g6  Nc6 bc  e5
[ECO B57 Sicilian: Sozin, not Scheveningen]
e4 c5  Nf3 d6  d4 cd  Nd4 Nf6  Nc3 Nc6  Bc4
[ECO B56 Sicilian]
e4 c5  Nf3 d6  d4 cd  Nd4 Nf6  Nc3 Nc6
[ECO B56 Sicilian: Venice attack]
e4 c5  Nf3 d6  d4 cd  Nd4 Nf6  Nc3 e5  Bb5
[ECO B56 Sicilian]
e4 c5  Nf3 d6  d4 cd  Nd4 Nf6  Nc3
```

Figura A.3 Porzione del file di testo contenente le varianti di apertura.

GnuChess viene distribuito con diversi file ASCII, contenenti libri di aperture di dimensioni diverse: da 80 a 12.000 linee di gioco.
La lettura del libro di apertura viene eseguita da GnuChess prima dell'inizio di ogni nuova partita. Il file viene letto e interpretato (cioè decodificata la notazione PGN nelle strutture dati interne del pro-

gramma), le posizioni incontrate durante l'interpretazione del file vengono memorizzate, opportunamente marcate come posizioni del libro delle aperture, nella tabella delle trasposizioni, insieme alla mossa successiva che il libro suggerisce. Se nel libro sono indicate più mosse per la stessa posizione, la scelta tra queste viene fatta in maniera casuale al momento di caricare la posizione nella tabella delle trasposizioni.
Durante la partita, GnuChess controlla se la posizione corrente è nella tabella delle trasposizioni marcata come posizione di apertura. Se è così viene giocata direttamente la mossa associata alla posizione stessa, altrimenti viene attivato l'algoritmo di ricerca.
Questa gestione del libro delle aperture è molto buona sotto l'aspetto della espandibilità, e anche della velocità di selezione della mossa in fase di gioco; il difetto principale sembra, invece, la necessità di interpretare interamente il file ASCII all'inizio di ogni partita, costringendo l'utente ad una attesa che può diventare snervante nel caso di libri di grandi dimensioni.

A.5 La funzione di valutazione

A.5.1 La funzione ExaminePosition

La funzione *ExaminePosition*( ) viene chiamata prima dell'inizio dell'algoritmo di ricerca ed ha il compito di esaminare la posizione alla radice dell'albero di gioco e settare di conseguenza molte variabili che contribuiranno alla valutazione statica delle posizioni che verranno incontrate durante la ricerca stessa. In particolare vengono inizializzate delle tabelle (*Mking*[64], *Mwpawn*[64], *Mbpawn*[64], *Mknight*[64], *Mbishop*[64]) che contengono parte del valore posizionale dei pezzi per ogni casa in cui possono venire a trovarsi durante l'analisi. Inoltre vengono inizializzate alcune variabili in funzione della variabile *stage* che indica lo stadio della partita in corso secondo la formula:

$$stage = \begin{cases} 0 & \text{se } tmtl > 3600 \\ \dfrac{3600 - tmtl}{220} & \text{se } tmtl \in [1400, 3600] \\ 10 & \text{se } tmtl < 1400 \end{cases}$$

dove con *tmtl* si indica la somma del materiale complessivo sulla scacchiera esclusi i Re e i Pedoni. In pratica stage assume valori da 0 (in fase di apertura) a 10 (in finali con pochi pezzi) (vedi fig. A.4). È da notare come un simile criterio di suddivisione delle fasi della partita sia indispensabile per un buon giocatore artificiale (ma anche umano), data la diversa rilevanza strategica dei pezzi in funzione dello stadio della partita; l'esempio più evidente è dato dal Re, che deve essere tenuto al sicuro in apertura e deve essere il più possibile centralizzato nel finale.

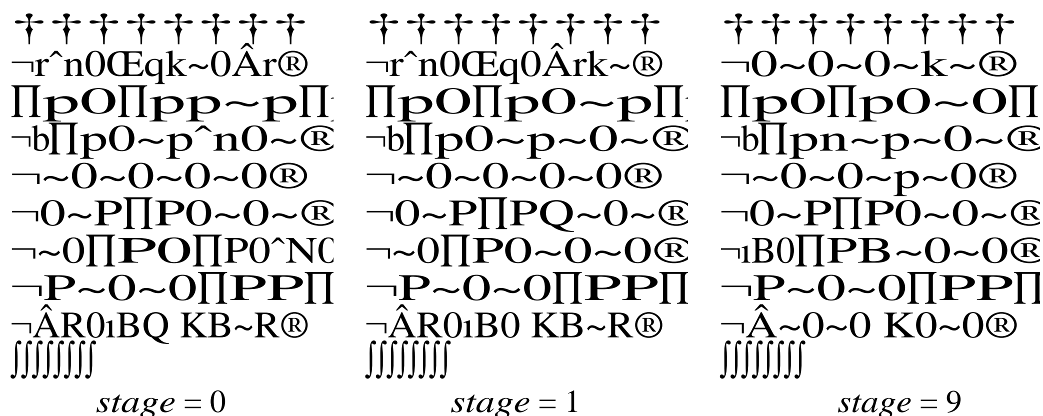


Figura A.4. Valore della variabile *stage* in diverse fasi della partita.

A.5.2 La funzione evaluate

La funzione che gestisce la valutazione statica è la *evaluate*(*short side,short ply,short alpha,short beta,short INCscore, short *InChk*) dove *side* codifica il giocatore dal cui punto di vista viene eseguita la valutazione, *ply* la profondità della posizione corrente nell'albero di gioco, *alpha* e *beta* sono i limiti inferiore e superiore usati dall'algoritmo AlphaBeta, *InChk* è un flag che viene settato quando c'è uno scacco. *INCscore* è una variabile usata per il calcolo incrementale della funzione di valutazione, in essa la funzione *evaluate* trova una stima del cambiamento del valore posizione che si è verificato con l'esecuzione dell'ultima mossa (ad es. se un pezzo è stato catturato il suo valore posizionale viene messo in *INCscore*, come pure eventuali variazioni alla struttura pedonale possono generare mutamenti al valore posizione e quindi registrati in *INCscore*). La prima operazione di *evaluate* è stimare la posizione con la formula:

$$s = -Pscore[ply\text{-}1] - INCscore + mtl[side] - mtl[xside]$$

dove l'array *Pscore*[*ply-1*] contiene il valore della posizione precedente all'ultima mossa analizzata escluso il materiale, e *mtl*[*side*] e *mtl*[*xside*] contengono rispettivamente il materiale complessivo sulla scacchiera per il giocatore *side* e per il suo avversario. La variabile *s* viene usata come risultato della funzione di valutazione solo se non è compresa nell'intervallo [*alpha* - 180,*beta* + 180] oppure nell'intervallo [*alpha* - 50,*beta* + 50] durante la ricerca quiescente, altrimenti viene chiamata la funzione *ScorePosition*(*side*) che effettua una valutazione statica della posizione. La variabile *s* non può essere sempre presa per buona a causa delle semplificazioni che vengono fatte dalla funzione *Makemove*(*mv*) durante il calcolo di *INCscore*. In particolare si può osservare che vengono presi in considerazione solo i mutamenti locali alla mossa *mv*, cioè alla casa di partenza e alla casa di arrivo del pezzo mosso, e che il valore posizionale del pezzo mosso non viene modificato a meno che non si tratti di una pedone. Comunque l'uso di *s* è utile quando è molto distante dalla finestra alpha-beta, per cui si presume che la posizione non rientri nella variante principale ed è quindi superfluo spendere altro tempo per una valutazione più accurata.

A.5.3 La funzione ScorePosition

ScorePosition(*side*) viene invocata quando è necessaria una valutazione statica accurata. Tutti i pezzi presenti sulla scacchiera vengono esaminati per attribuire a ciascuno il relativo valore posizionale a cui poi viene sommato il valore materiale. Il risultato di *ScorePosition* si può esprimere con la formula:

$$\begin{aligned} ScorePosition(side) = & \sum_{\forall piece\ p\ of\ side} positionalScore(p) - \sum_{\forall piece\ p\ of\ otherside} positionalScore(p) + \\ & + mtl[side] - mtl[otherside] + extra(side) - extra(otherside) \end{aligned}$$

La funzione *extra*(*side*) valuta alcuni aspetti della posizione che non possono essere osservati dalla funzione *positionalScore*(*p*). Le nozioni

scacchistiche che permettono a GnuChess di valutare una posizione possono essere suddivise in otto termini di conoscenza, seguendo la classificazione effettuata da Santi [San93]: materiale (m), buona sistemazione dei pezzi (b), mobilità e spazio (x), sicurezza del Re (k), controllo del centro (c), struttura pedonale (p), possibilità di attacco (a), relazione tra i pezzi (r).

Vediamo in dettaglio come questi termini di conoscenza contribuiscono alla valutazione di ciascun pezzo:

Pedone:

m: 100 punti per ogni Pedone sulla scacchiera;

b: Se è attaccato e non difeso penalità costante HUNGP, se è attaccato da più pezzi di quanti lo difendono penalità costante ATAKD.

x: Se il pedone è arretrato e non può essere mosso viene assegnata la penalità costante PBLOK.

k: Se il Re amico si trova a una distanza massima di due case e non è più nella casa iniziale (e1 per il Bianco, e8 per il Nero) e il pedone si trova nella colonna a, b, c, f, g o h, viene assegnato un bonus PAWNSHIELD(stage); se il Re amico è ancora nella casa iniziale e il pedone si trova nelle colonne a, b, g, h, nella traversa 2 o 3, allora viene assegnato il bonus PAWNSHIELD(*stage*)/2. In sintesi GnuChess ritiene più sicuro il Re è arroccato e ha dei pedoni vicini. PAWNSHIELD(*stage*) è decrescente, in quanto la sicurezza del Re diventa meno rilevante quando ci si avvicina al finale di partita.

c: Se il Pedone è nella colonna d o e, e non è ancora stato mosso, viene data una penalità pari a PEDRNK2B. Significato strategico: i pedoni centrali vanno spinti in apertura.

p: Ad ogni Pedone viene assegnato il punteggio PawAdvance[*sq*], dove *sq* è la casa occupata dal Pedone stesso, oppure 7/10 di PawAdvance[*sq*]-ISOLANI[column(*sq*)] se il Pedone è isolato. Se il Pedone è isolato viene assegnata la penalità costante PDOUBLED. Se il Pedone è arretrato viene assegnata la penalità dipendente dal numero dei pezzi avversari che lo attaccano BACKWARD[atkpiece], se inoltre la colonna in cui si trova il Pedone è semiaperta viene assegnata una ulteriore penalità PWEAK.

a: Se il Pedone si trova in una colonna semiaperta (priva di Pedoni avversari), viene valutata la possibilità che arrivi a promozione:

se l'avversario controlla con un Pedone una casa sul percorso di promozione del Pedone in esame, viene assegnato il bonus PassedPawn3(*stage,rank*), con *rank* si indica la traversa occupata dal Pedone; altrimenti se il Re nemico è nel quadrato di promozione oppure c'è un pezzo avversario in una casa sul percorso di promozione, allora viene assegnato il bonus PassedPawn3(*stage,rank*); se invece l'avversario ha almeno un pezzo che non sia il Re o un Pedone, viene assegnato il bonus PassedPawn1(*stage,rank*); se nessuna delle ipotesi precedenti è vera, viene assegnato il bonus PassedPawn0[*rank*] indipendente dalla variabile *stage*. Vale la relazione:

$$\text{PassedPawn0}[rank] \geq \text{PassedPawn1}(stage,rank) \geq \\ \geq \text{PassedPawn2}(stage,rank) \geq \text{PassedPawn3}(stage,rank)$$

per ogni $rank \in [0,7]$ e $stage \in [0,10]$ fissati.

Nelle posizioni di mediogioco ($stage < 5$) dove i Re si trovano su lati opposti della scacchiera, vengono premiati i Pedoni avanzati sul lato dove si trova il Re nemico, con 3 punti per ogni casa percorsa.

r: Viene assegnato PawnBonus(*stage*) per ogni Pedone. PawnBonus(*stage*) è crescente all'aumentare di *stage*, quindi viene premiata la presenza di molti Pedoni nel finale.

Cavallo:

m: 350 punti per ogni Cavallo.

b: Viene assegnato il valore KNIGHTSTRONG(*stage*), crescente verso il finale di partita, se il Cavallo occupa una casa forte, cioè non attaccabile da Pedoni avversari. Inoltre, se è attaccato e non difeso, o attaccato da pezzo di valore minore, subisce penalità costante HUNGP, se è attaccato da più pezzi di quanti lo difendono penalità costante ATAKD.

c: È assegnato un bonus dipendente dalla casa che occupa pknight[*sq*], definito in funzione del maggiore o minore controllo che il Cavallo esercita sulle case centrali della scacchiera.

k: Viene assegnato un bonus decrescente in funzione della distanza dal proprio Re, secondo il principio: più i pezzi sono vicini al proprio Re, più il Re è sicuro.

a: Viene assegnato un bonus decrescente in funzione della distanza dal Re nemico, secondo il principio: più i Cavalli sono vicini al Re

nemico, più il Re nemico è vulnerabile. Inoltre viene assegnato il bonus KNIGHTPOST(*stage*), crescente verso il finale di partita, per ogni pezzo avversario a distanza di 1 o 2 case dal Cavallo.

r: Se un giocatore ha ancora la coppia dei Cavalli riceve un bonus di 10 punti.

Alfiere:

m: GnuChess assegna 355 per ogni Alfiere, 5 punti in più rispetto ad un Cavallo.

b: Viene assegnato il valore BISHOPSTRONG(*stage*), crescente verso il finale di partita, se l'Alfiere occupa una casa forte, cioè non attaccabile da Pedoni avversari. Inoltre, se è attaccato e non difeso, o attaccato da pezzo di valore minore, subisce penalità costante HUNGP, se è attaccato da più pezzi di quanti lo difendono penalità costante ATAKD.

x: Viene assegnato un bonus, dipendente dal numero di case raggiungibili dall'Alfiere, BMBLTY[*numsq*].

c: Analogamente al caso del Cavallo, è assegnato un bonus dipendente dalla casa che occupa pbishp[*sq*], definito in funzione del maggiore o minore controllo che l'Alfiere esercita sulle case centrali della scacchiera.

a: Viene premiato l'attacco dell'Alfiere sulle case vicine al Re nemico con il punteggio KATAK per ogni casa attaccata. Viene premiato l'attacco di due pezzi avversari allineati, uno attaccato direttamente (che però non deve essere il Re o un Pedone) e l'altro "a raggi x", con il punteggio PINVAL (vedi fig. A.5a). Inoltre è anche premiata la situazione di attacco "a raggi x" anche se il pezzo attaccato direttamente è un pezzo amico, questa volta con il punteggio XRAY (fig. A.5b).

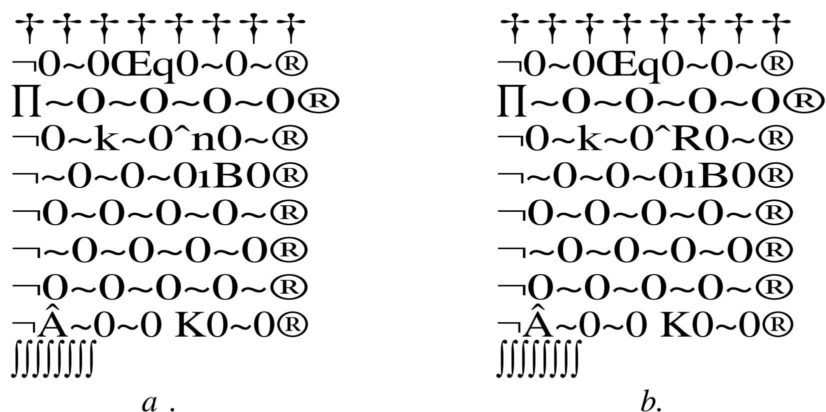


Figura A.5. Esempio di valutazione statica. a) L'Alfiere in g5 si merita il bonus PINVAL. b) In questo caso invece viene assegnato il bonus XRAY.

r: Se un giocatore ha ancora la coppia degli Alfieri riceve un bonus di 16 punti, rispetto ai 10 punti attribuitigli col possesso della coppia dei Cavalli. Concordemente alla attuale teoria scacchistica, GnuChess predilige la coppia degli Alfieri.

Torre:

m: 550 punti a ogni Torre sulla scacchiera.

b: Se è attaccata e non difesa, o attaccata da pezzo di valore minore, subisce penalità costante HUNGP, se è attaccato da più pezzi di quanti lo difendono penalità costante ATAKD.

x: Viene assegnato un bonus, dipendente dal numero di case raggiungibili dalla Torre, RMBLTY[*numsq*]. Inoltre se la Torre si trova su una colonna semiaperta (non occupata da Pedoni amici) guadagna il bonus RHOPN, se la colonna è completamente aperta (non ci sono neppure Pedoni avversari) guadagna anche il bonus RHOPNX.

a: Analogamente al caso dell'Alfiere, viene premiato l'attacco sulle case vicine al Re nemico con il punteggio KATAK per ogni casa attaccata. Viene premiato l'attacco di due pezzi avversari allineati, uno attaccato direttamente (che però non deve essere il Re o un Pedone) e l'altro "a raggi x", con il punteggio PINVAL. Ed è anche premiato l'attacco "a raggi x" anche se il pezzo attaccato direttamente è un pezzo amico, con il punteggio XRAY. Inoltre, se l'avversario ha ancora dei Pedoni, viene assegnato un bonus di 10 punti per la Torre in settima traversa. Una penalità proporzionale

alla distanza dal Re nemico viene assegnata a partire dal terzo stadio della partita (cioè con *stage* ≥ 3).

r: Per ogni Torre viene assegnato il bonus RookBonus(*stage*), crescente verso il finale della partita.

Donna:

m: 1100 punti per ogni Donna.

b: Se è attaccata e non difesa, o attaccata da pezzo di valore minore, subisce penalità costante HUNGP, se è attaccato da più pezzi di quanti lo difendono penalità costante ATAKD.

a: Viene assegnato un bonus di 12 punti se la distanza dal Re nemico è minore di 3 case. E, analogamente al caso della Torre, una penalità proporzionale alla distanza dal Re nemico viene assegnata se *stage* ≥ 3.

Re:

m: GnuChess assegna 1200 punti al Re, anche se ciò è irrilevante ai fini della valutazione della posizione, dato che i due re sono sempre presenti sulla scacchiera, e quindi i loro valori si annullano.

k: Come tutti sanno il Re va generalmente protetto fino al mediogioco e centralizzato nel finale. GnuChess applica questo principio strategico assegnando per il Re una combinazione lineare di due valori, KingOpening[*sq*] e KingEnding[*sq*], in maniera diversa a seconda dello stadio della partita: in apertura viene dato più peso al primo il primo, in finale al secondo. L'array KingOpening ha valori minimi nelle case centrali e massimi in prossimità degli angoli della scacchiera, per KingEnding vale il contrario. La combinazione di questi due valori viene ridotta di 1/2 se il materiale presente sulla scacchiera è maggiore di POSLIMIT. Se la colonna occupata dal Re è priva di Pedoni amici viene assegnata la penalità KHOPN(*stage*), se è priva di Pedoni nemici viene assegnata la penalità KHOPNX(*stage*), queste penalità vengono riassegnate se la colonna del bordo più vicino al Re è priva di Pedoni amici o nemici. Inoltre se l'avversario ha ancora la Donna e la colonna del bordo più vicino al Re non è occupata da Pedoni amici, viene assegnata la penalità AHOPEN pari a ben -200 punti. Se *stage* > 0, oppure lo sviluppo dei pezzi dell'avversario è stato completato, allora vengono assegnate penalità se il Re può essere minacciato da scacchi, o se non ci sono Pedoni vicino al Re. Se il Re è arroccato viene attribuito il bonus KCASTLD(*stage*), decre-

scente verso il finale, se invece è già stato mosso e non è arroccato si ha la penalità KMOVD(*stage*), anch'essa decrescente verso il finale di partita.

a: Viene assegnato una penalità proporzionale allo stadio della partita e alla distanza del Re sia ai propri Pedoni che ai Pedoni avversari; per favorire l'attacco del Re ai Pedoni passati avversari e il sostegno dei propri, la distanza viene considerata il quintuplo di quella reale. Questa penalità viene dimezzata se la somma di tutto il materiale presente sulla scacchiera, esclusi i Pedoni, è superiore alla soglia KINGPOSLIMIT.

Extra:

b: Viene assegnata la penalità costante HUNGX nel caso che contemporaneamente più di un pezzo di un colore sia attaccato e non difeso oppure attaccato da pezzo di minor valore.

a: Se un giocatore ha ottenuto uno *score* positivo (considerando tutti i termini di conoscenza sopra elencati) e non ha Pedoni, allora *score* viene posto a 0 se il giocatore in questione non ha per lo meno una Torre o una Donna o una coppia di pezzi minori, se ce l'ha subisce solo un dimezzamento dello *score* stesso se questo non è superiore al valore materiale di una Torre. Inoltre, se un giocatore ha almeno un Alfiere, o un pezzo di valore superiore, e l'avversario ha solo il Re, viene assegnato un bonus di 200 punti.

Vediamo un esempio di valutazione statica applicata alla posizione raffigurata nel seguente diagramma, in cui la mossa è al Bianco, di cui sono riportati i valori dei diversi fattori di conoscenza.

```
††††††††
¬0~0Ârr~k~®
¬~pŒqb~p∏p0
¬p~nıbO~n∏p
¬~0~p~0~0®
¬0~0ˆN0~0~®
¬~0~0ıBN∏PP
¬P∏PPŒQ0∏PB
¬~0~RÂR0 K0®
∫∫∫∫∫∫∫∫
```

m	0
b	0
x	4
k	20
c	-10
p	14
a	8
r	0

Figura A.6. Esempio di valutazione di una posizione.

Nella situazione illustrata in figura A.6 il materiale è pari (m=0) e nessun pezzo può essere molestato da una spinta di pedone (quindi b=0) e la mobilità e lo spazio sono leggermente a favore del Bianco (x=4). Anche la sicurezza del Re è favorevole al Bianco (k=20) a causa della maggiore vicinanza dei pezzi Bianchi al loro Re. Il controllo del centro segna 10 punti per il Nero a causa del controllo esercitato dal Pedone d5, che però, essendo isolato, fa perdere 14 punti al Nero nella valutazione che considera la struttura pedonale. Le possibilità di attacco sono considerate maggiori per il Bianco (a=8) principalmente per l'attacco a "raggi x" esercitato dalla Torre e1 sulla Torre e8 attraverso l'Alfiere e3. Infine si può osservare che nessun giocatore trae vantaggio da una migliore relazione tra i pezzi (r=0), dato che i due colori hanno i pezzi in egual numero e tipo. La valutazione complessiva, che si ottiene sommando le valutazioni parziali, dà un vantaggio al Bianco di 36 punti, in termini di materiale pari a poco più di 1/3 di Pedone.

A.6 Test su file di posizioni

Una utile caratteristica di GnuChess è la possibilità di eseguire test su insiemi di posizioni. Queste posizioni devono prima essere registrate in notazione Forsyth in un file ASCII.
La notazione Forsyth traduce la posizione in un formato testuale secondo le seguenti regole: la posizione viene trascritta per traverse partendo dalla 8 e finendo con la 1, al termine di ciascuna traversa viene posto il simbolo "/", i pezzi bianchi sono indicati con l'iniziale maiuscola e quelli neri con la minuscola, la quantità di case vuote tra due pezzi viene indicato con il relativo numero, fatta eccezione per i pezzi contigui. Alla fine della rappresentazione della scacchiera viene indicato il colore che ha la mossa con i simboli "w" e "b", e la mossa, o le mosse, in notazione algebrica, considerate corrette per la posizione in questione, eventualmente può anche essere presente un commento tra parentesi (fig. A.7).

††††††††
¬0~b~qÂr0~®
∏pr~0~k~0®
¬0∏p0~p^n0~®
¬~0~0ıBp~0®
¬0~p∏P0~0~®

¬~0∏P0~0~R®
¬P∏P0ŒQ0∏PB~®
¬ÂR0~0~0 K0®
∫∫∫∫∫∫∫∫

```
2b1qr2/pr3k2/1p2pn2/4Bp2/2pP4/2P4R/PP1Q1PB1/R5K1/w  Qd2g5
(Fischer,R - Sherwin,J,New Jersey 1957)
```

Figura A.7. Esempio di notazione Forsyth di una posizione di test.

A richiesta GnuChess è in grado di interpretare e analizzare le posizioni registrate nel file “Test Positions” e restituire il numero di mosse corrette che sono state trovate.

Appendice B

Deja Vu Chess Library

Deja Vu è un database di partite di Scacchi in formato Microsoft Fox Pro, distribuito su CD-ROM, leggibile sia da sistemi Macintosh che Windows. La versione che è stata usata per questa tesi è la 1.0 del 1994. In essa sono contenute 353.457 partite, giocate tra la fine del XVIII secolo e l'anno di uscita del CD-ROM stesso. Comunque la stragrande maggioranza delle partite risalgono alla seconda metà del nostro secolo. Le fonti di quasi tutte le partite sono tornei magistrali, campionati mondiali, zonali, nazionali ecc.; ciò, insieme al fatto di essere state giocate mediamente in un periodo recente, garantisce l'elevata qualità del gioco espresso nell'insieme del database. Non mancano partite di scarso valore tecnico, inserite solo come curiosità (per es. qualche partita di Napoleone), ma queste sono comunque in numero decisamente ininfluente rispetto alla mole di partite di livello professionistico.
Nel complesso, quindi, Deja Vu sembra essere una buona base di conoscenza, utilizzabile per l'apprendimento di un giocatore artificiale, sia per il numero statisticamente rilevante delle partite che per la loro qualità.

B.1 Formato del database

Il database è suddiviso in tre file: “dejavu.DBF”, “dejavu.FPT”, “dejavu.CDX”. I primi due contendono i dati, mentre il terzo contiene gli indici per un accesso più rapido ai dati durante le ricerche. Vediamo in dettaglio i formati dei file .DBF e .FPT.

Il file “dejavu.DBF” contiene informazioni circa i giocatori, l’anno, l’evento, il tipo di apertura in codice ECO, il numero di mosse e il risultato finale, mentre il file “dejavu.FPT” contiene la trascrizione delle partite, in notazione algebrica abbreviata, in forma testuale. I record dei due file sono messi in relazione uno a uno tramite un puntatore presente nel file .DBF (vedi fig. B.1).

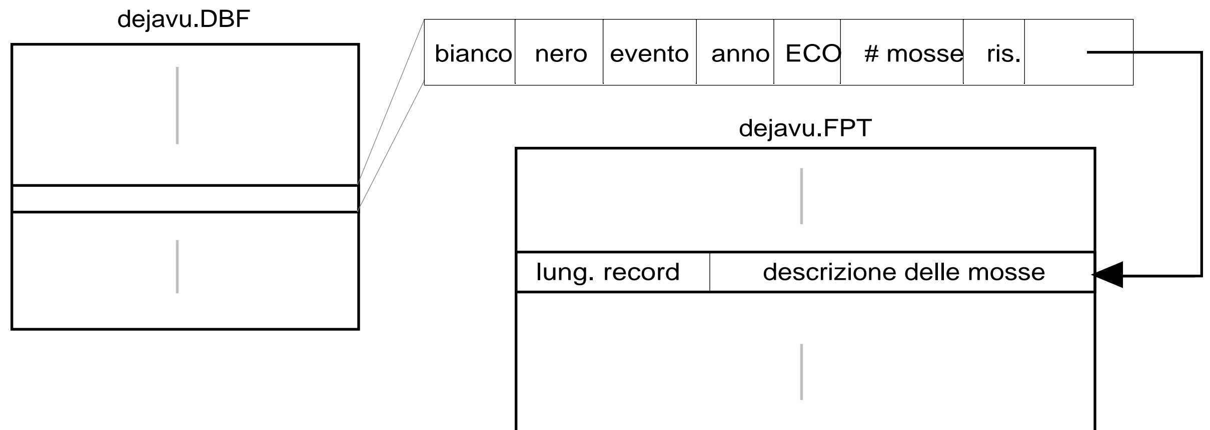


Figura B.1. Relazione tra i record di dejavu.DBF e quelli di dejavu.FPT.

I record del file .DBF sono di lunghezza fissa. Tutti campi sono in formato stringa, compreso il puntatore al corrispondente record in .DBF, nella tabella B.1 sono riportate le dimensioni dei campi. Al contrario, i record del file .FPT sono di lunghezza variabile. Il primo campo (un intero di 4 byte) indica la lunghezza in byte del secondo, che è in formato testo e riporta la trascrizione delle mosse della partita in notazione algebrica abbreviata (vedi fig. B.2).

Campo .DBF	Tipo	Byte
bianco	stringa	25
nero	stringa	25
evento	stringa	35
anno	stringa	4
ECO	stringa	3
numero di mosse	stringa	3
risultato	carattere	1
puntatore a .FPT	stringa	10
TOTALE		106

Campo .FPT	Tipo	Byte
lunghezza	intero	4
descrizione mosse	testo	64÷1280
TOTALE		68÷1284

Tabelle B.1 e B.2 - Record dei file dejavu.DBF e dejavu.FPT.

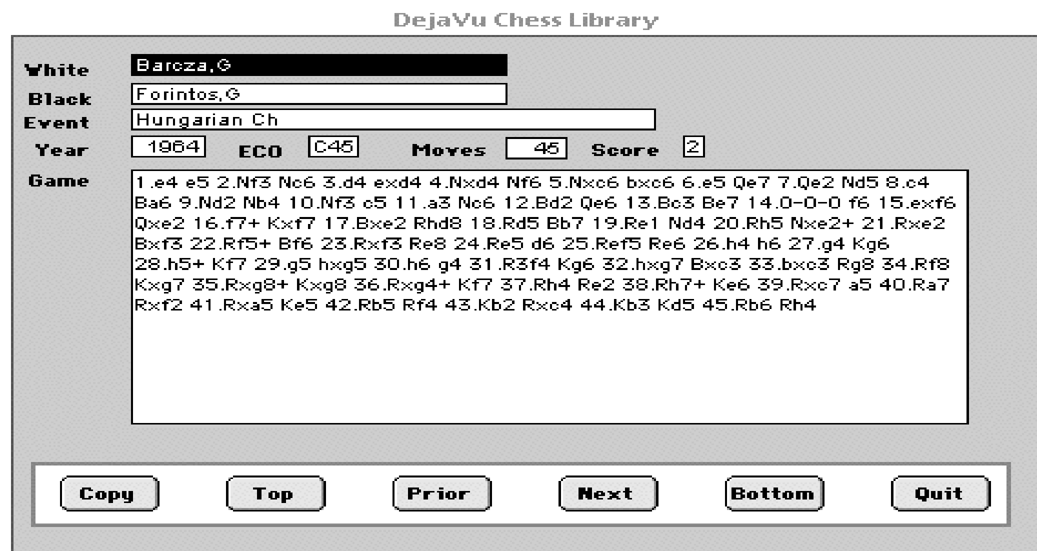


Figura B.2. Un record di Deja Vu.

B.2 Alcuni dati statistici

Per meglio valutare la validità complessiva del database sono stati rilevati alcuni dati statistici.
Nelle figure B.3 e B.4, è evidenziata la distribuzione delle partite nel corso degli anni. Si può osservare come il numero di partite per anno segua un andamento esponenziale rispetto al tempo. Questo ci può dare una indicazione approssimativa della qualità teorica delle partite (nel senso della teoria del gioco degli Scacchi): se assumiamo che le

partite più recenti abbiano mediamente un valore teorico superiore, possiamo affermare che il valore dell'intero database è circa uguale a quello di un database contenente solo partite molto recenti.
Nella tabella B.3 sono riportati i risultati riassuntivi delle partite presenti in Deja Vu, raggruppate per tipo di apertura. Confrontando questa tabella con quella analoga in [Sel95] costruita da un database di 215.000 partite giocate tra il 1980 al 1995, si osserva come i dati in generale non si discostino molto: in media la differenza non supera i 3 punti percentuali. Fanno eccezione aperture molto giocate in passato, come l'Italiana e il Gambetto di Re, e le aperture D00-D09, in cui è da registrare un incremento di 5÷6 punti dei risultati di patta; probabilmente cioè è dovuto allo sviluppo della teoria scacchistica, che è riuscita a neutralizzare varianti di apertura un tempo temibili. Per la Difesa Est-Indiana si può fare un raffronto anche con i dati riportati in [Pon89] (pag. 210), riguardanti circa 2000 partite degli anni '80 giocate in tornei di VII categoria FIDE e superiori (media ELO >2400). In queste partite si registra un numero assai superiore di patte, ben il 45% contro il 32% di Deja Vu, ma la percentuale di punti ottenuti dai due colori non differisce di molto: 58,5% per il Bianco in [Pon89], 57% per il Bianco in Deja Vu. Probabilmente il numero così alto di patte rilevate da Ponzetto è dovuto al tipo di tornei scelti per la statistica, nei quali le patte si verificano spesso, a volte anche dopo poche mosse.
Infine dalla tabella B.4 si può notare come i giocatori più ricorrenti nel database siano tra i più forti di questo secolo. Un indice di valutazione abbastanza preciso per misurare il livello di gioco delle partite sarebbe stato, soprattutto per quelle degli ultimi anni, il punteggio Elo dei giocatori, ma questo dato non è stato inserito in Deja Vu.

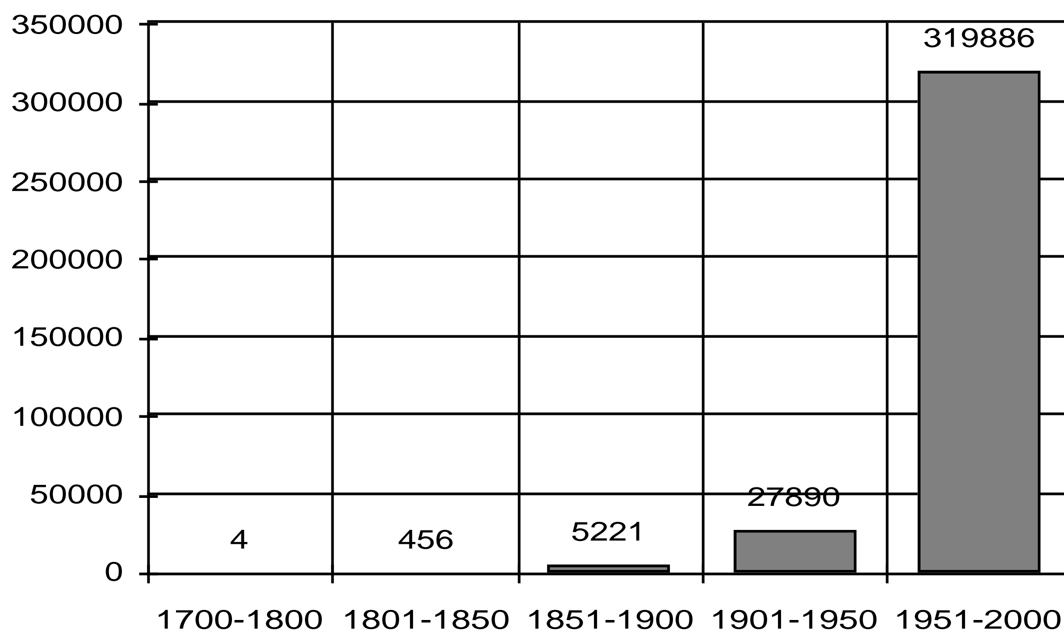


Figura B.3. Distribuzione temporale delle partite in Deja Vu.

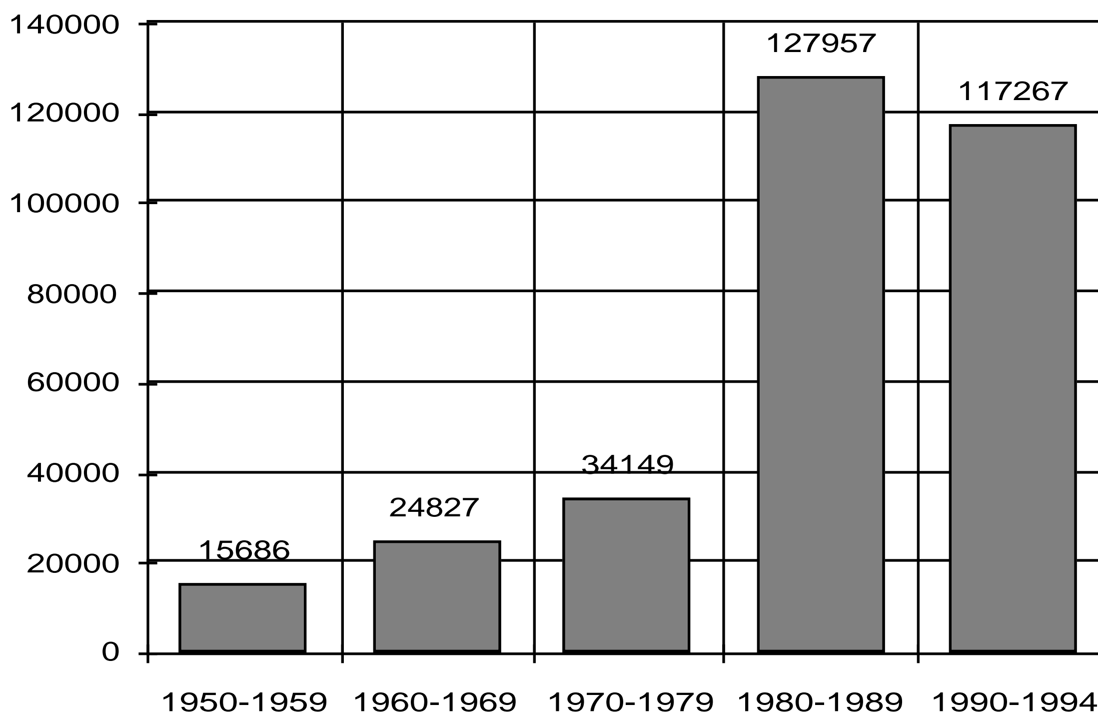


Figura B.4. Distribuzione temporale delle partite in Deja Vu dal 1950 in poi.

Apertura	Codici ECO	Giocate	1-0	Patta	0-1
Larsen, Bird, 1.Cf3, ecc...	A00-A09	12833	36	32	32
Inglese	A10-A39	30225	36	38	26
Torre, Tromposvky, ecc...	A40-A49	14858	37	33	30
Gambetto Benko	A50-A59	8530	42	30	28
Moderna Benoni	A60-A79	2405	43	28	29
Olandese	A80-A99	7632	41	33	26
Pirc, Scandinava, Alekhin, ecc...	B00-B09	20557	41	31	28
Caro Kann	B10-B19	7115	41	34	25
Siciliana - varianti minori	B20-B29	2235	39	29	32
Siciliana - Sveshnikov	B30-B39	8952	40	34	26
Siciliana - Paulsen, Taimanov	B40-B49	9905	36	34	30
Siciliana - varie	B50-B59	4232	33	35	33
Siciliana - Rauzer	B60-B69	3795	38	35	27
Siciliana - Dragone	B70-B79	5505	40	30	30
Siciliana - Scheveningen	B80-B89	11885	39	31	30
Siciliana - Najdorf	B90-B99	7040	40	31	29
Francese	C00-C19	22174	41	32	27
1. e4 e5 varie	C20-C29	3197	41	25	34
Gambetto di Re	C30-C39	3636	46	16	38
Quattro Cavalli e simili	C40-C49	10632	41	32	27
Italiana e simili	C50-C59	6572	38	26	36
Spagnola	C60-C99	24799	38	36	26
1. d4 varie	D00-D09	11368	42	28	30
Slava e simile	D10-D19	6200	35	44	21
Gambetto di Donna accettato	D20-D29	3563	40	37	23
Gambetto di Donna rifiutato	D30-D69	28070	39	40	21
Grünfeld	D70-D99	24163	35	43	22
Catalana	E00-E09	5144	38	42	20
Ovest e Bogo-indiana	E10-E19	13951	34	44	22
Nimzo-indiana	E20-E59	13004	35	37	28
Est-indiana	E60-E99	26816	41	32	27
TOTALI	A00-E99	353457	39	34	27

Tabella B.3. Statistica dei risultati delle partite in Dejavu raggruppate per aperture.

Giocatore	Part
Korchnoi,V	2829
Tal,M	2504
Portisch,L	1990
Smyslov,V	1952
Gligoric,S	1943
Timman,J	1925
Petrosian,T	1833
Karpov,A	1806
Hort,V	1699
Larsen,B	1687
Geller,E	1681
Spassky,B	1606
Ivkov,B	1540
Alekhine,A	1528
Miles,A	1499
Bronstein,D	1493
Polugaevsky,L	1459
Keres,P	1443
Ljubojevic,L	1367
Uhlmann,W	1313
Andersson,U	1242
Euwe,M	1225
Beliavsky,A	1218
Romanishin,O	1189
Tukmakov,V	1180
Gheorghiu,F	1171
Najdorf,M	1163
Sax,G	1157
Kasparov,G	1142
Vaganian,R	1128
Reshevsky,S	1112
Farago,I	1091
Ribli,Z	1085
Huebner,R	1059
Adorjan,A	1052
Browne,W	1051
Botvinnik,M	1048
Jansa,V	1031
Szabo,L	1031
Tartakower,S	1012
Gulko,B	1005
Csom,I	1001
Van der Wiel,J	1001
Nunn,J	992
Balashov,Y	973
Short,N	973
Psakhis,L	964
Smejkal,J	944
Taimanov,M	936
Donner,J	922
Seirawan,Y	905
Yusupov,A	904
Westerinen,H	903
Chandler,M	897
Ftacnik,L	886
Speelman,J	884
Fischer,R	865
Lobron,E	865
Matulovic,M	851
Nikolic,P	849
Panno,O	849
Spielmann,R	848
Christiansen,L	839
Tseshkovsky,V	822
Lukacs,P	792
Ree,H	789
Velimirovic,D	784
Kindermann,S	781
Dolmatov,S	779
Marshall,F	773
Benko,P	764
Sveshnikov,E	763
Tarrasch,S	760
Kavalek,L	759
Petursson,M	758
Capablanca,J	751
Colias,B	750
Pachman,L	750
Sosonko,G	749
Adams,M	743
Rossetto,H	743
Sokolov,A	730
Kholmov,R	729
Unzicker,W	725
Chernin,A	724
Ehlvest,J	724
Suba,M	724
Van der Sterren,P	719
Torre,E	718
Lputian,S	715
Rogers,I	712
Malaniuk,V	710
Gufeld,E	703
Razuvaev,Y	699
Suetin,A	697
Vasiukov,E	687
Bogoljubow,E	686
Kupreichik,V	680
Knaak,R	679
Gurevich,M	675
Fedorowicz,J	673
Stahlberg,G	671
Piket,J	670
Khalifman,A	668
Kurajica,B	667
Rubinstein,A	667
Bagirov,V	662
Schmidt,W	658
Yudasin,L	650
Zapata,A	649
Benjamin,J	646
Hodgson,J	643
Savon,V	642
Bisguier,A	639
Hjartarson,J	638
Nogueiras,J	638
Rodriguez,A	638
Ivanovic,B	636
Kuzmin,G	632
Anand,V	631
Forintos,G	629
Plachetka,J	628
Rashkovsky,N	626
Hebden,M	624
Flohr,S	622
Alburt,L	621
Eliskases,E	619
Maroczy,G	615
Ivanchuk,V	613
Gurevich,D	605
Vogt,L	603
Lerner,K	601
Eingorn,V	600
Marjanovic,S	597

Tabella B.4. I giocatori di cui sono riportate più partite in Deja Vu.

Bibliografia

[AklNew77] AKL S.G., NEWBORN M.M., "The Principle Continuation and the Killer Heuristic", *ACM Annual Conference*, 1977, 466-473.

[Ana90] ANANTHARAMAN T.S., *A Statistical Study of Selective Min-max Search in Computer Chess*, PhD. Thesis, Carnegie Mellon University, Pittsburgh, 1990.

[Bag67] BAGLEY J.D., "The Behaviour of Adaptive Systems which Employ Genetic and Correlation Algorithms", in *Dissertation abstracts international*, vol. 28, n. 12, 1967.

[Bal92] BALDI, P., *Calcolo delle probabilità e statistica*, MacGraw-Hill, 1992.

[Bal94] BALKENHOL, B., "Data Compression in Encoding Chess Positions", in *International Computer Chess Association Journal*, vol. 17, n. 3, settembre 1994, 132-140.

[Bea92] BEAL, D., "The 1992 QMW Uniform-Platform Autoplay Computer-Chess Tournament", in *International Computer Chess Association Journal*, vol. 15, n. 3, settembre 1992, 162-167.

[Bea93] BEAL, D., "The 1993 QMW Uniform-Platform Autoplay Computer-Chess Championship", in *International Computer Chess Association Journal*, vol. 16, n. 3, settembre 1993, 166-170.

[Bea95] BEAL, D., "The 8th World Computer-Chess Champioship. Round-by-Round", in *International Computer Chess Association Journal*, vol. 18, n. 2, giugno 1995, 94-94.

[BEGC90] BERLINER H. J., EBELING C., GOETSCH G., CAMPBELL M. S., "Measuring the Performance Potential of Chess Programs", in H.J.Berliner et al., *Artificial Intelligence*, n. 43, 1990, 7-20.

[Bot82] BOTVINNIK M., "Decision Making and Computers", in *Advances in Computer Chess 3*, (M. R. B. Clarke ed.), Pergamon Press, Oxford, 1982, 169-179.

[BUV94] BREUKER D.M., UITERWIJK J.W.H.M., VAN DEN HERIK H.J., *Replacement Schemes for Transposition Tables*, in *International Computer Chess Association Journal*, vol. 17, n. 4, dicembre 1994, 183-193.

[CKSSTF88] CHEESEMAN P., KELLY, J., SELF, M., STUTZ, J., TAYLOR, W., FREEMAN D., "Autoclass: A Bayesian Classification System", in *Proceedings of the Fifth International Workshop on Machine Learning*, Morgan Kaufmann, San Mateo, CA, 1988, 296-306.

[Cia92] CIANCARINI P., *Giocatori artificiali* , Mursia 1992.

[Cia94a] CIANCARINI P., "Esperiments in Distributing and Coordinating Knowledge", in *International Computer Chess Association Journal*, vol. 17, n. 3, 1994, 115-131.

[Cia94b] CIANCARINI P., "Distributed Searches: a Basis for Comparison", in *International Computer Chess Association Journal*, vol. 17, n. 4, 1994, 194-206.

[Elo86] ELO A.E., *The Rating of Chessplayers Past and Present*, second edition, Arco Publishing Inc., New York, 1986.

[GeoSch88] GEORGE M., SCHAEFFER J., *Chunking for Experience*, Department of Computing Science, University of Alberta, 1988.

[Ghe93] GHERRITY M., *A Game-Learning Machine*, PhD. Thesis, University of California, San Diego, 1993.

[Gil78] GILLOGLY J.J., *Performance Analysis of the Technology Chess Program*, Department of Computer Science, Carnegie-Mellon University, 1978.

[Gol89] GOLDBERG D.E., *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison-Wesley, 1989.

[Har87a] HARTMANN D., "Hox to Extract Relevant Knowledge from Grandmasters Games. Part 1: Grandmasters Have Insights – the Problem is What to Incorporate into Pactical Programs",

in *International Computer Chess Association Journal*, vol. 10, n. 1, marzo 1987, 14-36.

[Har87b] HARTMANN D., “Hox to Extract Relevant Knowledge from Grandmasters Games. Part 2: The Notion of Mobility, and the Work of De Groot and Slater”, in *International Computer Chess Association Journal*, vol. 10, n. 2, giugno 1987, 78-90.

[Har89] HARTMANN D., “Notions of Evaluation Functions Tested against Grandmaster Games”, in *Advances in Computer Chess 5* (ed. D.F. Beal), Elsevier Science Publishers B. V., 1989, 91-141.

[Hol75] HOLLAND J.H., *Adaptation in Natural and Artificial Systems*, The University of Michigan Press, 1975.

[Hut94] HUTCHINSON A., *Algorithmic Learning*, Clarendon Press Oxford, 1994.

[Kai90] KAINDL K., “Tree Searching Algorithms”, in Marsland T.A., Schaeffer J., *Computers, Chess, and Cognition*, Springer-Verlang, New York, 1990, 133-158.

[InsMac] “Apple Event Manager”, in *Inside Macintosh*, vol. VI, cap. 6.

[LeeMah90] LEE K., MAHAJAN S., “The Development of a World Class Othello Program”, in *Artificial Intelligence*, n. 43, 1990, 21-36.

[Lev91] LEVINSON R., *Experience-Based Creativity*, Department of Computer and Information Sciences University of California Santa Cruz, 1991.

[Levy91] LEVY D., *How Computers Play Chess*, Computer Science Press, New York 1991.

[Meu89] VAN DEN MEULEN M., “Weight Assessment in Evaluation Functions”, in *Advances in Computer Chess 5* (ed. D.F. Beal), Elsevier Science Publishers B. V., 1989, 81-89.

[Mug88] MUGGLETON S.H., “Inductive Acquisition of Chess Strategies”, in Michie D., Hayes J.E. e Richards J., *Machine Intelligence 11*, Oxford University Press, 1988, 375-389.

[Nik89] NIKOLAICZUK L., *Gezielte Mittelspielstrategie: 100 x Französisch*, Thomas Beyer Verlags GmbH, Hollfeld, 1989.

[Nit82] NITSCHE T., “A Learning Chess Program”, in *Advances in Computer Chess 3*, Pergamon Press, Oxford, 1982, 113-120.

[Pon89] PONZETTO P., *La Difesa Est-Indiana*, Mursia, 1989.

[Sam59] SAMUEL A., "Some Studies in Machine Learning using the Game of Checkers", in *IBM Journal of Research and Development*, 3, 1959, 210-229.

[San89] SANDSTROEM H. E., *New move Generation Algoritm*, file distribuito con la documentazione di GnuChess, 1989.

[San93] SANTI A., *Un programma a conoscenza distribuita per il gioco degli scacchi*, Tesi di Laurea, Università degli Studi di Bologna, Corso di Laurea in Scienze dell'Informazione, a.a. 1992/93.

[Sch83] SCHAEFFER J., "The History Heuristic"", in *International Computer Chess Association Journal*, vol. 6, n. 3, 1983, 16-19.

[Sch86] SCHAEFFER J., *Experiments in Search and Knowledge*, PhD. Thesis, University of Alberta, 1986.

[SST90] SCHERZER T., SCHERZER, L., TJADEN, D., "Learning in Bebe", in *Computers, Chess, and Cognition* (eds. Marsland, T.A., Schaeffer, J.), Springer-Verlang, New York 1990, 197-216.

[Sel95] SELLERI C., "Il database dà i numeri", in *Torre & Cavallo*, anno XI, n. 4, aprile 1995, 21-22.

[Sha50] SHANNON C.E., "Programming a Computer for Playing Chess", in *Philosophical Magazine*, vol. 41 (7th series), Taylor & Francis Ltd., 1950, 256-275. Ripubblicato in *Computer Chess Compendium* (ed. David Levy), B.T. Batsford, London 1988.

[Sim83] SIMON R., "Why Should Machines Learn?", in *Machine Learning: and Artificial Intelligence Approach* (eds. Michalski R., Carbonall J., Mitchell T.), Tioga, Palo Alto California, 1983.

[Sla87] SLATE D.J., "A Chess Program that Uses its Transposition Table to Learn from Experience", *International Computer Chess Association Journal*, vol. 10, n. 2, 1987, 59-71.

[Tak91] TAKVORIAN Y., *Echess et C, Initiation à l'analyse et à la programmation du jeu d'échecs*, Editions Radio, Paris 1991.

[Tes88] TESAURO G., "Connectionist Learning of Expert Backgammon Evaluations", in *Machine Learning* (ed. Laird, J.), Ann Arbor, Michigan 1988, 200-206.

[Thr95] THRUN S., "Learning To Play the Game of Chess", in *Neural Information Processing System 7* (eds. Tesauro, G., Touretzky, D., Leen, T.), 1995.

[TigHer91] VAN TIGGELEN A., VAN DEN HERIK H.J., "ALEXS: An Optimization Approach for the Endgame KNNKP(h)". in *Advances in Computer Chess 6*, Ellis Hoorwood Ltd., Chichester, 1991, 161-177.

[Toz93] TOZZI M., *Progetto e realizzazione di un programma distribuito di visita di alberi di gioco*, Tesi di Laurea, Università degli Studi di Pisa, Corso di Laurea in Scienze dell'Informazione, a.a. 1992/93.

[Tun91] TUNSTALL-PEDOE W., "Genetic Algorithms Optimizing Evaluation Functions", in *International Computer Chess Association Journal*, vol. 14, n. 3, settembre 1991, 119-128.

[Tur53] TURING A., "Chess", in *Faster Than Throught* (ed. B.V. Bowden), London, Pitman 1953, 286-295. Ripubblicato in *Computer Chess Compendium* (ed. David Levy), B.T. Batsford, London, 1988.

[Zob70] ZOBRIST A. L., *A New Hashing Method with Application for Game Playing*, Technical Report n. 88, Computer Science Department, The University of Wisconsin, Madison 1970, ristampato in *International Computer Chess Association Journal*, vol. 13, n. 2, giugno 1990, 69-74.