Contenuto

Capitolo 1

Introduzione

1.1 Giocatori artificiali

Ha sempre suscitato un grande interesse la possibilità di affidare ad un calcolatore il compito di prendere decisioni su problemi di carattere strategico, in cui vi siano da effettuare scelte sulla base di considerazioni di carattere generale, difficilmente riconducibili a formule matematiche. Nel corso degli ultimi anni si è sviluppata una branca della scienza, l'Intelligenza Artificiale, che si occupa appunto di costruire macchine e strumenti capaci di compiere autonomamente questo tipo di scelte.
Un ambiente in cui si è notevolmente sviluppato lo studio dell'Intelligenza Artificiale è quello dei giochi, ed in particolare di quei giochi in cui i partecipanti sono due, conoscono tutte le possibili continuazioni in ogni momento della partita ed ottengono con le loro mosse vantaggi equivalenti agli svantaggi dell'avversario. L'esempio più evidente di ciò è rappresentato dal gioco degli scacchi: attualmente, infatti, esistono in commercio diversi calcolatori in grado di giocare ad un livello paragonabile con quello dei migliori giocatori umani.
Chiameremo giocatore artificiale un sistema in grado di ricoprire autonomamente in questo tipo di giochi il ruolo di uno dei due partecipanti, con il compito di operare delle scelte mirate all'ottenimento di un vantaggio nei confronti dell'avversario.

1.2 Ricerca o conoscenza?

Le componenti fondamentali che un sistema "intelligente" usa per il raggiungimento dei suoi obiettivi sono la ricerca e la conoscenza: la prima consente di passare in rassegna un certo numero di decisioni ammissibili, mentre la seconda permette di determinarne la qualità. Per un giocatore artificiale, la ricerca è intesa come lo sviluppo delle linee di gioco future, mentre la conoscenza è l'insieme di nozioni, tratte dal dominio di applicazione, che da una parte consentono di scegliere quali linee di gioco sviluppare, e dall'altra

permettono di assegnare loro un valore.
La valutazione di una situazione del gioco è il tentativo di indicare quanto essa sia favorevole in considerazione delle prospettive future. Non essendo generalmente possibile l'analisi di tutte le continuazioni permesse, un giocatore (sia umano che artificiale) fonda il proprio giudizio sulla conoscenza del dominio di applicazione e sulla esplorazione delle situazioni raggiungibili. Esiste una sorta di bilanciamento tra queste due componenti, nel senso che maggiore conoscenza possiede un giocatore, e minore necessità ha di valutare le continuazioni future (essendo in grado di esprimere un giudizio sulla base di elementi contingenti), e viceversa una maggiore capacità di esplorazione delle linee di gioco future compensa una limitata conoscenza.
Data l'enorme quantità di conoscenza che richiedono i giochi più complessi, e data la difficoltà della realizzazione di un sistema esperto anche solo per semplici problemi, ciò che si è verificato negli ultimi anni è stato lo sviluppo di giocatori artificiali in cui la profondità di ricerca ha ricevuto un'attenzione prevalente. Il miglioramento delle prestazioni è stato ottenuto fornendo ai giocatori artificiali la capacità di esaminare più continuazioni, e più in profondità: questo è derivato inizialmente dall'uso di migliori algoritmi di ricerca e di macchine sempre più veloci, e successivamente dall'introduzione di algoritmi di ricerca paralleli.
La parallelizzazione della ricerca consiste nella suddivisione tra più processori del compito di scandagliare le continuazioni possibili, con un vantaggio, rispetto alla ricerca sequenziale, nella quantità di continuazioni esaminate. Il guadagno derivato dall'utilizzo di più unità di elaborazione autonome non è tuttavia illimitato: tra i motivi di degrado c'è la necessità di coordinazione e sincronizzazione dei processori, oltre alla dispersione delle informazioni inerenti la ricerca.
Esiste dunque un limite al miglioramento che si può ottenere ponendo l'attenzione sull'aumento della profondità di ricerca, e questo ha portato alcuni autori all'esame di nuovi approcci. Ad esempio, in [26] prima, ed in [3] poi, si è cercato, sia pure con esperimenti differenti, di valutare il comportamento di un giocatore artificiale di scacchi che si serve di più unità di elaborazione in maniera innovativa. In pratica, ogni unità effettua la valutazione delle linee di gioco future rispetto ad una propria specifica conoscenza del dominio di applicazione, e propone la continuazione che ritiene migliore; dalle mosse proposte viene poi estratta quella definitiva da comunicare all'avversario. Tale approccio è stato successivamente ripreso in [29]: il metodo descritto e sperimentato in questo lavoro consiste nella distribuzione su più processori della conoscenza tratta da uno stesso programma di scacchi sequenziale. Il giocatore artificiale risultante è così composto da istanze del programma originale che si differenziano soltanto per la porzione della conoscenza che è stata loro associata. Anche in questo caso ogni istanza sceglie la mossa migliore dal suo punto di vista, e dalle mosse selezionate viene in seguito estratta quella definitiva.

1.3 Obiettivi del lavoro

Col presente lavoro ci proponiamo di esaminare in maniera più approfondita il metodo di distribuzione della conoscenza, consistente nella parallelizzazione di un programma mediante l'assegnamento di porzioni della sua conoscenza a più unità di elaborazione: ognuna di esse provvederà alla scelta di una continuazione in base alla conoscenza che le è stata assegnata, e dalle continuazioni così determinate verrà estratta, con un opportuno criterio di selezione, quella definitiva da comunicare all'avversario.
In particolare, studieremo la distribuzione della conoscenza in un giocatore artificiale di scacchi, realizzato con il linguaggio di coordinazione Network C-Linda riusando il codice di un programma sequenziale di scacchi di dominio pubblico (Gnuchess 4.0). Il programma ottenuto, denominato Gnupar, consentirà di distribuire secondo varie combinazioni i termini di conoscenza ricavati dal programma originale, e sarà in grado di applicare diversi criteri di selezione della mossa finale. A causa del grande numero di possibili istanziazioni di Gnupar, verranno eseguiti esperimenti (su un campione di 500 posizioni) tendenti a determinare quali di esse garantiscano le prestazioni potenzialmente migliori: su queste verrà eseguita la sperimentazione finale, consistente nella disputa di tornei di 20 partite contro il programma sequenziale originale, ed in alcuni casi nell'esecuzione di singole partite contro giocatori umani (verrà utilizzata l'interfaccia grafica Xboard, distribuita insieme a Gnuchess 4.0). Grazie ad un'apposita versione di Xboard, saranno anche disputate partite contro utenti dell'Internet Chess Server.
Infine, si tenterà di migliorare le prestazioni di Gnupar intervenendo, con l'applicazione di un algoritmo genetico in un caso e con modifiche intuitive in un altro, nella regolazione dei criteri di selezione della mossa finale.
La sperimentazione verrà eseguita su un'architettura parallela composta da una rete locale di 11 SUN SparcStation collegate in Ethernet. In particolare, si rileva che due workstation risultano più veloci delle altre; di questo si terrà debito conto nella definizione degli esperimenti e nella valutazione dei risultati: si cercherà infatti, per avere valutazioni uniformi, di evitare il loro utilizzo in tutti quegli esperimenti in cui avesse rilevanza il tempo di risposta.
Gli esiti del nostro lavoro saranno considerati positivi se verrà trovata almeno un'istanziazione di Gnupar che si dimostri sostanzialmente più forte del programma sequenziale da cui è stato tratto.

1.4 Struttura della tesi

Nel prossimo capitolo esamineremo innanzitutto la base teorica ed i principali algoritmi sequenziali che si possono usare per la realizzazione di un giocatore artificiale, evidenziando alcune euristiche introdotte per il miglioramento delle prestazioni. Successivamente presenteremo alcuni dei principali metodi

esistenti per la parallelizzazione della ricerca, ed introdurremo i principi e le motivazioni che portano alla definizione della distribuzione della conoscenza come nuovo approccio alla parallelizzazione di un giocatore artificiale. Illustreremo quindi le possibilità offerte e le esperienze precedenti relativamente allo specifico campo d'interesse che intendiamo analizzare (gli scacchi), ed infine descriveremo il modello di programmazione parallela (Linda) di cui ci serviremo per la realizzazione di Gnupar.
Il capitolo 3 ha per argomento la descrizione del programma sequenziale di scacchi Gnuchess 4.0, di cui riuseremo il codice. Questo capitolo si articolerà attraverso l'analisi delle componenti che gestiscono le attività di Gnuchess 4.0, con particolare riferimento alla funzione di valutazione, dalla quale sarà estratta la conoscenza da distribuire all'interno del giocatore artificiale che intendiamo realizzare. Verrà inoltre confrontato Gnuchess 4.0 con la versione 3.1 dello stesso programma, usata precedentemente in [29], e sarà infine eseguita una descrizione dell'interfaccia grafica (Xboard) che si intende utilizzare per la disputa delle partite nella sperimentazione conclusiva.
Il capitolo 4 concerne l'analisi e la sperimentazione di alcuni criteri di distribuzione della conoscenza e di selezione della mossa finale. Data la molteplicità di giocatori diversi che potrebbero scaturire dalla combinazione dei criteri considerati, tenteremo di determinare da quali possono derivare i giocatori con le prestazioni potenzialmente migliori eseguendo alcuni esperimenti su un insieme di 500 posizioni scacchistiche.
Nel capitolo 5 descriveremo il giocatore artificiale con distribuzione della conoscenza che intendiamo realizzare: verrà evidenziata per esso l'architettura generale, oltre al protocollo di comunicazione che regolerà la coordinazione tra i vari processori.
Infine, nel capitolo 6 sarà eseguita la sperimentazione conclusiva, limitata ad alcuni giocatori che nel capitolo 4 hanno dimostrato di avere le migliori potenzialità. Verranno anche compiuti alcuni tentativi per la determinazione di giocatori in grado di ottenere prestazioni migliori; in particolare, si cercherà di intervenire sulla definizione di due criteri di selezione: in uno attraverso l'uso di un algoritmo genetico, e nell'altro intuitivamente.

1.5 Ringraziamenti

Si desidera ringraziare la Free Software Foundation per la concessione dei programmi Gnuchess 4.0 ed Xboard, e Tim Mann per avere fornito la versione di Xboard in grado di connettere Gnuchess 4.0 con l'Internet Chess Server. Un particolare ringraziamento va inoltre a Mirco Tozzi, di cui abbiamo pesantemente utilizzato il lavoro per quanto riguarda sia l'impostazione che la realizzazione e la sperimentazione di Gnupar.

Capitolo 2

Programmi di gioco

2.1 Giochi a informazione completa

La teoria dei giochi prevede lo studio di problemi decisionali che coinvolgono in generale più giocatori in competizione l'uno con l'altro, aventi interessi differenti e miranti ciascuno ad ottimizzare il proprio profitto [31]. In particolare, l'attenzione degli studiosi si è concentrata su una sottoclasse particolare di problemi decisionali, le cui caratteristiche sono :

1. i partecipanti sono due;

2. ciò che un partecipante perde equivale al guadagno dell'altro (0-sum);

3. tutte le decisioni possibili sono note ad entrambi i partecipanti in ogni momento (informazione perfetta).

Esempi di problemi decisionali di questo tipo sono i giochi degli scacchi e della dama.
Il modello matematico discreto su cui è basata la teoria relativa ai particolari problemi che soddisfano queste condizioni è rappresentato da una struttura ad albero, detta albero di gioco [28], in cui i nodi identificano stati del gioco, e gli archi identificano quelle azioni, ammesse dalle regole del gioco, che causano la trasformazione dallo stato del nodo padre a quello del nodo figlio. L'insieme degli archi uscenti da un nodo dell'albero di gioco rappresenta tutte e sole le decisioni che sono ammesse nella situazione rappresentata dal nodo. Un nodo terminale dell'albero di gioco rappresenta uno stato particolare del gioco in cui non esistono decisioni legali, e ad esso è quindi associato un valore che sintetizza l'esito finale del gioco (ad esempio, nei giochi come gli scacchi questo valore deve identificare la vittoria del bianco, la vittoria del nero o il pareggio).

2.2 Realizzazione sequenziale

2.2.1 MiniMax

Consideriamo un qualsiasi gioco appartenente alla sottoclasse di problemi decisionali descritta: come può un giocatore decidere quale mossa gli conviene scegliere ad un determinato punto del gioco?
Chiaramente, egli deve seguire una strategia che gli consenta di arrivare allo stato terminale a cui è associato il valore più vantaggioso. L' esempio più importante di strategia di controllo per questa classe di giochi è rappresentato dall'algoritmo MiniMax [1].
L'algoritmo MiniMax realizza una brutale ricerca sul sottoalbero di gioco che ha come radice il nodo rappresentante la situazione attuale, ed esegue un esame esaustivo di tutte le possibili sequenze di mosse fino ai nodi terminali dell'albero. Da questo esame scaturisce una valutazione dei nodi figli del nodo attuale, valutazione che permette al giocatore di effettuare una scelta (chiaramente verrà scelta la mossa che porta al nodo figlio a cui è associata la valutazione più vantaggiosa per il giocatore a cui tocca muovere). Le ipotesi su cui si basa l'algoritmo sono proprio quelle che contraddistinguono questa sottoclasse di problemi della teoria dei giochi, e cioè:

- quella di informazione perfetta, a seguito della quale in ogni momento entrambi i giocatori sono in grado di selezionare la mossa migliore secondo il proprio punto di vista;
- quella di 0-sum, a seguito della quale il vantaggio di un giocatore corrisponde allo svantaggio dell'altro, per cui il nodo di valore maggiore per uno è allo stesso tempo il nodo di valore minore per l'altro.

Sintetizzando, un giocatore deve scegliere la mossa che porta al nodo di valore maggiore quando considera posizioni in cui tocca a lui muovere, e la mossa che porta al nodo di valore minore quando considera posizioni in cui tocca all'avversario (sempre considerando il suo punto di vista nella valutazione dei nodi).
Vediamo una descrizione dell'algoritmo MiniMax, secondo una sintassi C-like.

---

input: il nodo da valutare e il giocatore a cui tocca muovere;
output: il valore del nodo per il giocatore di riferimento player1;

```
value
MiniMax (nodo, gioc)
status nodo;
color gioc;
{
```

```
value val[MAXSONS];

if (nodo e' una foglia)
    return (valore_per_player1(nodo));
else
    {
    for (ogni figlio x[i] di nodo)
        val[i] = MiniMax (x[i], other[gioc]);

    if (gioc == player1)
        return ( max (val[]) );
    else
        return ( min (val[]) );
    }
}
```

2.2.2 NegaMax

Una variante di MiniMax più semplice dal punto di vista della codifica è detta NegaMax [20]. Differisce solo per il fatto che la valutazione dei nodi è eseguita rispetto al giocatore a cui tocca muovere in quel momento, e non rispetto sempre allo stesso giocatore. L'algoritmo NegaMax è il seguente:

---

input: il nodo da valutare;
output: il valore del nodo per il giocatore a cui tocca muovere;

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

if (nodo e' una foglia)
    return (valore_per_me(nodo));
else
    {
    for (ogni figlio x[i] di nodo)
        val[i] = - NegaMax (x[i]);
    return ( max (val[]) );
    }
}
```

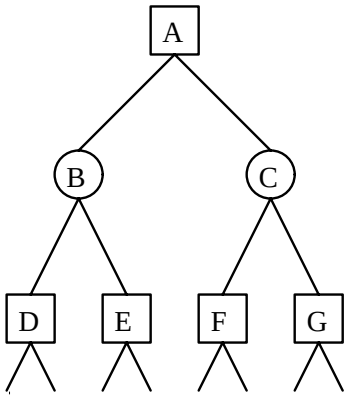

Figura 2.1: Albero da visitare

L'equivalenza dei due algoritmi deriva dal fatto che se un nodo ha valore $val(nodo, player1)$ per un giocatore, e $val(nodo, player2)$ per il suo avversario, allora è $val(nodo, player1) = -val(nodo, player2)$, per cui

$$\min_i(val(x_i, player1)) = \min_i(-val(x_i, player2)) = -\max_i(val(x_i, player2))$$

2.2.3 AlphaBeta

La valutazione del valore di un nodo secondo la strategia MiniMax è eseguita generalmente con una visita a scandaglio (depth-first), facilmente realizzabile con una procedura ricorsiva, la quale richiede un modesto spazio di memoria, proporzionale alla profondità di ricerca. Ma invece di visitare completamente l'albero di gioco, come fa l'algoritmo MiniMax, è possibile ottenere lo stesso risultato scartando quei sottoalberi che risultano ininfluenti per la determinazione del valore finale. L'algoritmo classico che realizza questo miglioramento è detto AlphaBeta [13], e si basa sulle seguenti considerazioni.

Supponiamo per semplicità di dover visitare un albero con un fattore 2 di diramazione, come quello in figura 2.1.
Supponiamo inoltre di eseguire l'algoritmo MiniMax rispettando un ordinamento da sinistra a destra ogni volta che si esaminano i figli di un nodo.
La valutazione comincia con l'esecuzione di MiniMax(A, player1), e siccome A non è una foglia, allora viene determinato prima val[B] = MiniMax(B, player2), che supponiamo produca il valore $valB$, e poi val[C] = MiniMax(C, player2), che supponiamo produca il valore $valC$; infine viene calcolato il valore finale $valA = max\{valB, valC\}$.
Se per ipotesi α è un limite inferiore del valore della radice A dell'albero (questo è sicuramente vero se si pone $\alpha = -\infty$), allora sarà $\alpha \leq valA$, quindi $\alpha \leq max\{valB, valC\}$. Chiaramente, essendo questo valido solamente per il max, non è detto che sia vero per i valori che non rappresentano il max, quindi può ugualmente essere $\alpha > valB$ oppure $\alpha > valC$.

Dopo aver trovato il valore $valB$ del primo figlio considerato, si hanno 3 possibilità:

- $valB < \alpha$: siamo in presenza di un valore che sicuramente non determina il valore massimo;
- $valB = \alpha$: non si può dire niente;
- $valB > \alpha$: siamo in presenza di un nuovo limite inferiore del valore della radice A. Infatti è $valB \leq max\{valB, valC\} = valA$, per cui anche $valB$ limita inferiormente il valore della radice A, e per di più è un limite più stringente di α. Ponendo $\alpha = valB$ prima di andare a valutare il figlio C, si fa avvicinare il valore di α a quello della radice senza cambiare i termini del problema.

In generale, ripetendo questi controlli ordinatamente per ogni figlio della radice, si ottengono man mano delle valutazioni effettive che da una parte consentono di avvicinare sempre più il limite inferiore α al valore della radice (quando la valutazione ricavata è maggiore del limite inferiore attuale), e dall'altra permettono di scoprire quei nodi che non influiranno nella successiva determinazione del valore massimo (quando la valutazione ricavata è minore del limite inferiore attuale).
Ritorniamo ora all'esecuzione dell'algoritmo MiniMax, e sviluppiamo la determinazione del valore dei figli, che era stata lasciata indeterminata. Quando viene eseguito MiniMax(B, player2), siccome B non è una foglia viene determinato prima val[D] = MiniMax(D, player1), che supponiamo produca il valore $valD$, e poi val[E] = MiniMax(E, player1), che supponiamo produca il valore $valE$; infine viene calcolato $valB = min\{valD, valE\}$.
Dopo aver trovato il valore $valD$, potrebbe accadere che $valD < \alpha$: essendo $valB \leq valD$ per definizione, allora anche $valB < \alpha$, e quindi per quanto detto prima si è già sicuri che il nodo B ha un valore che non sarà preso in considerazione per la determinazione del valore del nodo A. In tal caso diventa inutile proseguire questa istanza dell'algoritmo MiniMax, e si può interrompere a questo punto la valutazione di questo ramo dell'albero saltando l'esame dei restanti figli di B. In tale circostanza si dice che viene eseguito un taglio (cut-off).

Generalizzando (vedere figura 2.2), se α è un limite inferiore del valore della radice (a profondità 0), allora si possono valutare ordinatamente tutti i figli della radice (a profondità 1), facendo man mano tendere α al valore effettivo della radice; il limite inferiore α così determinato può causare dei tagli nella valutazione dei figli dei figli della radice (a profondità 2), nel senso che appena si trova un valore minore di α si può interrompere la valutazione del figlio della radice in cui ciò si è verificato (essendo esso sicuramente ininfluente nella determinazione del valore della radice), e passare alla valutazione

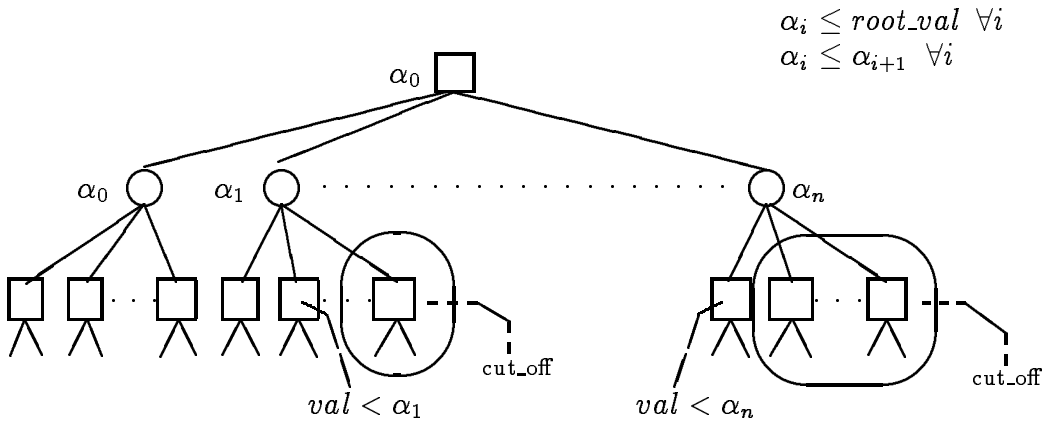

Figura 2.2: Limite inferiore α

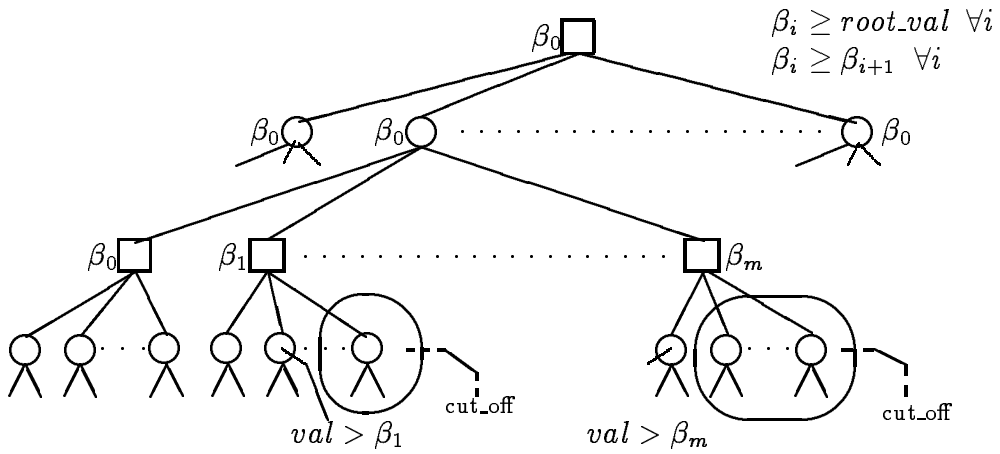

Figura 2.3: Limite superiore β

del figlio successivo della radice.

Supponiamo ora che β sia un limite superiore del valore della radice A (questo è sicuramente vero se si pone $\beta = +\infty$): sarà quindi $\beta \geq valA = max\{valB, valC\}$, per cui $\beta \geq valB$ e $\beta \geq valC$.

Generalizzando (vedere figura 2.3), se β è un limite superiore del valore della radice (a profondità 0), allora β è anche un limite superiore al valore di tutti i figli della radice (a profondità 1). Considerando singolarmente ciascun figlio x_i della radice, si ottiene, analogamente ad α, che quando si valutano ordinatamente i figli di x_i (a profondità 2), si fa man mano tendere β al valore effettivo di x_i ; il limite superiore β così determinato può causare dei tagli nella valutazione dei figli dei figli di x_i (a profondità 3), nel senso che appena si trova un valore maggiore di β si può interrompere la valutazione del figlio di x_i in cui ciò si è verificato (essendo esso sicuramente ininfluente nella determinazione del valore di x_i), e passare alla valutazione del figlio

successivo di x_i.

Unendo le due considerazioni fatte per α e per β, si ottiene un algoritmo che riguarda direttamente una porzione di albero di gioco profonda 3, in cui grazie ad α si possono eseguire dei tagli al livello 2 e grazie a β si possono eseguire dei tagli al livello 3. Indirettamente, invece, viene interessato tutto l'albero, in quanto l'algoritmo è chiamato ricorsivamente e quindi i limiti inferiore e superiore vengono propagati lungo i rami.
I due parametri α e β ($\alpha \leq root\_val \leq \beta$) definiscono in pratica un intervallo $[\alpha, \beta]$ detto $\alpha\beta$-window che serve per l'esclusione di sottoalberi ininfluenti durante la visita dell'albero di gioco.
Si ha il seguente algoritmo.

---

input: il nodo da valutare, il giocatore a cui tocca muovere e gli estremi della finestra $\alpha\beta$;
output: il valore del nodo per il giocatore di riferimento player1;

```
value
AlphaBeta_MnMx (nodo, gioc, alpha, beta)
status nodo;
color gioc;
value alpha, beta;
{
value val[MAXSONS];

if (nodo e' una foglia)
  return (valore_per_player1(nodo));
else
  if (gioc == player1)
    {
    for (ogni figlio x[i] di nodo)
      {
      val[i] = AlphaBeta_MnMx (x[i], other[gioc], alpha, beta);

      if (val[i]>alpha) /* miglioramento del limite inferiore */
        alpha = val[i];

      if (val[i]>beta)           /* esecuzione di un taglio: */
        return (+INFINITO);      /* il valore di nodo non e' */
      }                          /*  sicuramente il minimo */

    return (alpha);  /* al termine e' alpha = max (val[]) */
    }
  else
```

```
    {
    for (ogni figlio x[i] di nodo)
      {
      val[i] = AlphaBeta_MnMx (x[i], other[gioc], alpha, beta);

      if (val[i]<alpha)          /* esecuzione di un taglio: */
        return (-INFINITO);      /* il valore di nodo non e' */
                                 /*  sicuramente il massimo */

      if (val[i]<beta)  /* miglioramento del limite superiore*/
        beta = val[i];
      }

    return (beta);  /* al termine e' beta = min (val[]) */
    }
}
```

Anche in questo caso come per l'algoritmo NegaMax, se si considera per ogni livello dell'albero il punto di vista del giocatore a cui tocca muovere, si ha una versione semplificata: infatti, invertendo ad ogni cambiamento di livello il segno dei valori ed i sensi delle disuguaglianze, si ha una equivalenza tra le azioni da eseguire nei livelli pari e dispari dell'albero.
Ne risulta il seguente algoritmo.

---

input: il nodo da valutare e gli estremi della finestra $\alpha\beta$;
output: il valore del nodo per il giocatore a cui tocca muovere;

```
value
AlphaBeta_NgMx (nodo, alpha, beta)
status nodo;
value alpha, beta;
{
value val[MAXSONS];

if (nodo e' una foglia)
  return (valore_per_me(nodo));
else
  {
  for (ogni figlio x[i] di nodo)
    {
    val[i] = - AlphaBeta_NgMx (x[i], -beta, -alpha);
```

```
    if (val[i]>alpha) /* miglioramento del limite inferiore */
      alpha = val[i];

    if (val[i]>beta)  /* esecuzione di un taglio */
      return (+INFINITO);
    }

  return (alpha);  /* al termine e' alpha = max (val[]) */
  }
}
```

L' efficienza dell'algoritmo AlphaBeta è tanto migliore quanto prima vengono valutati i figli che determinano i tagli, quindi è fortemente dipendente dall'ordinamento di valutazione dei figli. Inoltre, quanto più piccola e' la finestra $\alpha\beta$ e tanto più probabile è l'esecuzione dei tagli, fermo restando che il valore della radice deve essere contenuto in essa.

2.2.4 Aspiration Search

L'algoritmo detto Aspiration Search è stato definito per cercare di ridurre le dimensioni della finestra $\alpha\beta$ con cui eseguire la visita dell'albero di gioco. Esso esegue una valutazione provvisoria V della radice, tramite qualche funzione di valutazione apposita, e pone quindi $\alpha = V - \epsilon$ e $\beta = V + \epsilon$ (invece della canonica $\alpha = -\infty$ e $\beta = +\infty$); se il valore risultante esce dalla finestra così definita, deve essere ripetuta la valutazione spostando la finestra dalla parte in cui è uscito il valore. È chiaro che le prestazioni dipendono fortemente dalla capacità di previsione della finestra $\alpha\beta$ iniziale.
L' algoritmo in versione NegaMax è il seguente:

---

input: il nodo da valutare;
output: il valore del nodo per il giocatore a cui tocca muovere;

```
value
AspirationSearch (nodo)
status nodo;
{
value initval, firstval, alpha, beta;

initval = evaluate (nodo);
alpha = initval - EPS;
beta = initval + EPS;
```

```
firstval = AlphaBeta_NgMx (nodo, alpha, beta);

if (firstval<alpha)
     return (AlphaBeta_NgMx (nodo, -INFINITO, alpha));
else
     if (firstval>beta)
         return (AlphaBeta_NgMx (nodo, beta, INFINITO));
     else
         return (firstval);
}
```

2.2.5 Variante Fail Soft

Le prestazioni di Aspiration Search sono condizionate dalla possibilità di fallimento della prima esecuzione di AlphaBeta: esiste una variante dell'algoritmo AlphaBeta che, in caso di fallimento, ritorna un valore il quale definisce un limite più stringente per la finestra della ricerca successiva nell' Aspiration Search.
Questa variante è detta Fail Soft AlphaBeta, e nella versione NegaMax è così definita:

---

input: il nodo da valutare e gli estremi della finestra $\alpha\beta$;
output: il valore del nodo per il giocatore a cui tocca muovere;

```
value
F_AlphaBeta_NgMx (nodo, alpha, beta)
status nodo;
value alpha, beta;
{
value val[MAXSONS];
value best;

if (nodo e' una foglia)
  return (valore_per_me(nodo));
else
  {
  best = -INFINITO;

  for (ogni figlio x[i] di nodo)
    {
    val[i] = - F_AlphaBeta_NgMx (x[i], -beta, -max(alpha, best));
```

```
    if (val[i]>best) /* miglioramento del limite inferiore */
      best = val[i];

    if (val[i]>beta)  /* esecuzione di un taglio */
      return (best);
    }

  return (best);  /* al termine e' best = max (val[]) */
  }
}
```

Il valore ritornato dalla variante Fail Soft dell'algoritmo AlphaBeta ha le seguenti caratteristiche:

1. se è compreso nella finestra $\alpha\beta$ è il valore corretto del nodo esaminato;
2. se è minore di *alpha* è un limite superiore del valore corretto del nodo esaminato;
3. se è maggiore di *beta* è un limite inferiore del valore corretto del nodo esaminato;

A questo punto, l'algoritmo di Aspiration Search diventa il seguente.

---

input: il nodo da valutare;
output: il valore del nodo per il giocatore a cui tocca muovere;

```
value
F_AspirationSearch (nodo)
status nodo;
{
value initval, firstval, alpha, beta;

initval = evaluate (nodo);
alpha = initval - EPS;
beta = initval + EPS;

firstval = F_AlphaBeta_NgMx (nodo, alpha, beta);

if (firstval<alpha)
     return (F_AlphaBeta_NgMx (nodo, -INFINITO, firstval));
else
     if (firstval>beta)
         return (F_AlphaBeta_NgMx (nodo, firstval, INFINITO));
```

```
    else
        return (firstval);
}
```

2.2.6 Albero Ridotto

La maggior parte dei giochi più interessanti genera alberi di gioco le cui dimensioni sono troppo grandi per consentire una visita completa. Anche supponendo di utilizzare l'algoritmo AlphaBeta in condizioni ottimali, visitando sempre per primi i nodi che causano dei tagli (la porzione di albero visitata viene detta in questo caso albero minimale), spesso non è sufficiente neanche il più veloce degli elaboratori per ottenere una risposta in tempi accettabili. Per questo motivo l'analisi deve necessariamente essere limitata ad un sottoalbero (detto albero ridotto) con la stessa radice dell'albero da valutare, ma troncato in qualche maniera. Le foglie dell'albero ridotto possono anche non essere nodi terminali dell'albero di gioco; ad esse viene perciò associato un insieme di informazioni proprie del gioco in oggetto, da utilizzare per ricavare un valore approssimato attraverso una specifica funzione di valutazione statica.
Come deve essere eseguita la scelta dell'albero ridotto?
Un metodo comune è quello di fissare una profondità di ricerca (detta orizzonte) e di troncare l'albero di gioco a questa profondità. È chiaro che l'incapacità di conoscere i possibili sviluppi oltre l'orizzonte può portare a grossolani equivoci nella valutazione: in particolare possono esistere alcuni nodi che più di altri sono fonte di errore nel caso di loro valutazione statica. Gli stati di gioco corrispondenti a questi nodi sono detti posizioni instabili (non quiescenti), e la loro caratterizzazione dipende fortemente dal tipo di gioco in esame (ad esempio, negli scacchi sono instabili le posizioni che occorrono dopo la cattura di un pezzo o dopo lo scacco al Re).
Risulta evidente la necessità di evitare che le posizioni instabili siano tra i nodi terminali dell'albero ridotto, ed è quindi opportuno che in concomitanza di queste posizioni sia data la possibilità di superare la profondità limite, possibilmente fino ad arrivare a posizioni stabili: questo ampliamento viene detto ricerca quiescente.

2.2.7 Euristiche di miglioramento sequenziale

Vista la impossibilità di visita completa dell'albero di gioco per la maggior parte dei problemi che ci interessano, d'ora in avanti considereremo unicamente l'esame di un albero ridotto scelto opportunamente, supponendo di possedere una funzione di valutazione statica in grado di fornire un valore sufficientemente approssimativo di quei nodi che sono non terminali per il

primo e terminali per il secondo. Fatta questa precisazione, andiamo ad esaminare alcune euristiche utili per migliorare l'efficienza dell'algoritmo AlphaBeta, delle quali si può trovare una discussione in [11].

1. TABELLA DELLE TRASPOSIZIONI: durante la ricerca in un albero, la medesima posizione può presentarsi più volte: per evitare di valutarla nuovamente la si memorizza in una tabella da cui potrà poi essere recuperata in qualsiasi momento. Per identificare l'elemento della tabella in cui sono contenuti i dati di una determinata posizione viene utilizzata una funzione hash, la quale ha la peculiarità di non essere bijettiva: per questo motivo potrebbero anche esserci più posizioni che fanno capo allo stesso elemento della tabella (collisioni).
   Le informazioni contenute di solito in un elemento della tabella sono:

   - un identificatore della posizione: serve per sapere se la posizione memorizzata è effettivamente quella da valutare o se invece si tratta di una collisione;
   - la valutazione statica della posizione;
   - il giocatore rispetto a cui è valutata la posizione;
   - la profondità a cui si è arrivati nella visita dell'albero durante l'esecuzione della valutazione.

   Generalmente, si memorizza una posizione solo se la sua valutazione deriva dalla ricerca in un sottoalbero con profondità rilevante; inoltre, se due posizioni diverse fanno riferimento alla stessa entry della tabella, viene memorizzata quella valutata con costo computazionale maggiore (cioè la cui valutazione statica è arrivata ad una profondità maggiore dell'albero).
   Alla tabella delle trasposizioni si accede durante la visita dell'albero di gioco per vedere se in essa è già contenuta la valutazione richiesta, oppure alla fine di una visita che abbia raggiunto una profondità sufficiente, per memorizzare le informazioni elaborate.

2. EURISTICA DEI KILLER: viene denominata killer una mossa che ha prodotto un taglio durante la ricerca AlphaBeta. Si ipotizza che una mossa che ha prodotto un taglio rispetto alla valutazione di una mossa ad un certo livello possa produrre dei tagli anche per altre mosse sempre allo stesso livello; questo è valido in particolare per quelle posizioni in cui solo poche tra le mosse possibili sono in grado di evitare una minaccia: quando la minaccia provoca il taglio di una mossa che non la sventa, essa viene memorizzata e sarà valutata tra le prime in risposta alle altre mosse possibili.
   Normalmente, per ogni livello dell'albero viene gestita una breve lista di mosse killer.

3. TABELLA DELLE CONFUTAZIONI: vengono memorizzate le mosse che confutano le alternative alla continuazione principale.

4. TABELLA HISTORY: l'idea è che una mossa risultata buona in una posizione può risultare buona anche per altre posizioni in cui è legale: si memorizzano le mosse risultate migliori così da esaminarle per prime in una nuova posizione in cui sono legali.

5. APPROFONDIMENTO ITERATIVO: si valuta l'albero fino ad una profondità d, dopodiché si utilizzano le informazioni raccolte per valutarlo una seconda volta a profondità $d+1$, e poi si ripete incrementando ogni volta la profondità di ricerca.
   Le informazioni che agevolano la visita successiva sono:

   - ordinamento delle mosse: dopo ogni iterazione, la lista delle mosse legali del nodo radice viene riordinata in modo che le mosse risultate migliori in una ricerca siano esaminate per prime nella iterazione successiva, con un ovvio vantaggio nella esecuzione dei tagli. Siccome tutte le mosse che formano la variante principale di una ricerca saranno esaminate per prime nella ricerca successiva, si ha una buona probabilità di esaminare subito la mossa che produce un taglio.
   - valore della posizione: il valore risultante dalla ricerca a profondità d potrà essere utilizzato come centro della aspiration-window nella ricerca successiva a profondità $d+1$.
   - statistiche varie: per poter applicare le euristiche precedenti nella ricerca successiva, si ha la disponibilità di informazioni varie ricavate nell'iterazione appena conclusa, come le valutazioni statiche di certe posizioni, o quali mosse hanno prodotto un taglio, ecc.

2.3 Parallelizzazione della ricerca

Ci sono diversi punti durante la ricerca nell'albero di gioco in cui è possibile distribuire su più processori l'esecuzione delle varie routine, con conseguente guadagno di tempo e quindi di efficienza.
Vediamo una breve rassegna di possibili approcci al processo di parallelizzazione della ricerca sequenziale [5].

1. Parallelizzazione delle attività che riguardano i singoli nodi: ad esempio, si può distribuire su più processori l'operazione di valutazione statica delle foglie dell'albero ridotto; si possono suddividere tra più processori compiti di generazione dei nodi successivi (nel caso degli scacchi , si può assegnare ogni colonna della scacchiera ad un processore diverso, con il compito di generare le mosse dei pezzi presenti solamente in essa);

2. Ricerca con finestre parallele [6]: si può distribuire l'algoritmo Aspiration Search su N processori. Si divide l'intervallo $]-\infty,+\infty[$ in N sottointervalli $]-\infty,x_1[$, $[x_1,x_2[$, ..., $[x_{n-2},x_{n-1}[$, $[x_{n-1},+\infty[$, ognuno dei quali costituirà la finestra di ricerca associata ad un determinato processore. Gli N processori eseguiranno simultaneamente l'algoritmo Alfa-Beta rispetto alla finestra associata, ed al termine uno solo di essi troverà la soluzione (quello che ricaverà un valore compreso all'interno della sua finestra). Il tempo per arrivare alla soluzione sarà minore rispetto all'usuale algoritmo AlphaBeta, perché prima di tutto ogni istanza avrà una finestra di dimensioni ridotte, quindi eseguirà un maggior numero di tagli, ed inoltre non ci sarà bisogno di una successiva ricerca nel caso che il valore trovato non sia all'interno della finestra iniziale, come avviene nell'Aspiration Search sequenziale.

3. Mapping dello spazio di ricerca nei processori [29]: una funzione hash associa i nodi dell'albero di gioco a determinati processori. Quando un processore deve valutare un nodo, invia ai processori associati ai figli la richiesta di valutazione, e così via fino alle foglie dell'albero ridotto, le quali verranno valutate direttamente dai processori associati ad esse. Il risultato della valutazione statica verrà ritornato al processore che l'ha richiesto, e così via fino al nodo radice. In questo modo una determinata posizione sarà sempre valutata staticamente dallo stesso processore, per cui l'utilizzo di una transposition table locale ad ogni processore raggiunge un risultato ottimale (la valutazione statica viene eseguita una sola volta sia all'interno di un processore che rispetto anche agli altri processori);

4. Decomposizione dell'albero di gioco (Tree-Splitting): si divide l'albero in sottoalberi, e si distribuiscono i vari sottoalberi tra i processori. Ogni volta che un processore determina il valore della radice del sottoalbero che gli è stato associato, lo comunica al processore a cui è associato il padre di questa sottoradice, il quale provvede a paragonarlo con i valori degli altri figli, così da ricavare il valore che sarà poi passato ricorsivamente al processore opportuno.
   Esistono diversi algoritmi per questo metodo di parallelizzazione della ricerca: i principali sono Mandatory-Work-First [2] e Principal-Variation-Split [22, 24].

Il primo metodo può sempre essere realizzato, ed anzi nel caso degli scacchi esistono già versioni di giocatori artificiali che utilizzano componenti hardware specializzate in questo (come per esempio HITECH [12]).
Per la valutazione degli altri metodi è necessario considerare i seguenti motivi di degrado [23]:

- overhead di comunicazione: è il carico addizionale che un programma parallelo deve sopportare quando è impiegato tempo non trascurabile per la comunicazione di messaggi tra i processi;

- overhead di ricerca: è dato dal fatto che in ambiente distribuito le informazioni ricavate fino ad un dato punto della ricerca sono disperse su processori diversi, e quindi non completamente utilizzabili: ciò può portare ad una esplorazione non necessaria di una porzione dell'albero di gioco, con conseguente crescita delle sue dimensioni, e quindi con perdita di efficienza;

- overhead di sincronizzazione: è il costo che occorre quando alcuni dei processori sono inattivi in attesa che altri abbiano terminato il loro lavoro, vuoi perché stanno aspettando da essi dati senza i quali non potrebbero proseguire, vuoi perché hanno bisogno di un oggetto condiviso accedibile in mutua esclusione ed al momento occupato.

Utilizziamo il rapporto tra il tempo richiesto dalla esecuzione sequenziale ed il tempo richiesto da quella parallela (speedup) come parametro per la valutazione del miglioramento.
Nel metodo di ricerca con finestre parallele è notevole l'overhead di ricerca, mentre sono trascurabili gli altri due tipi di overhead; numerosi esperimenti hanno dimostrato che lo speedup è limitato da un fattore 5 o 6, indipendentemente dal numero di processori [6].
Nel metodo di mapping dello spazio di ricerca nei processori l'overhead di comunicazione tra i processori può diventare insostenibile [15].
Nel metodo di decomposizione dell'albero di gioco, anche ammettendo di utilizzare sempre un algoritmo che realizzi il migliore bilanciamento tra le forme di degrado, si ha che il guadagno in termini di speedup tende a stabilizzarsi, e a diventare, con l'aumentare del numero N dei processori, proporzionale a $\sqrt{N}$ [16].

2.4 Distribuzione della conoscenza

Allo stato dell'arte esiste quindi un limite per il numero di processori impiegati in un algoritmo di ricerca parallela oltre il quale non si ottiene alcun beneficio dall'aumento di risorse di elaborazione. Per cercare di aggirare l'ostacolo, alcuni autori hanno intrapreso nuove strade: ad esempio in [3] ed in [26] sono descritti due esperimenti tendenti a valutare le prestazioni di giocatori artificiali di scacchi composti da unità di elaborazione diverse che esplorano autonomamente l'albero di gioco e forniscono un insieme di mosse da cui viene estratta quella definitiva. Questo approccio suggerisce l'idea di eseguire la parallelizzazione di un programma di gioco distribuendo su più unità di elaborazione la sua conoscenza piuttosto che i suoi compiti di visita dell'albero di gioco. I vantaggi derivati dalla distribuzione della conoscenza si basano sulle considerazioni che seguono.
Il costo computazionale di ogni decisione si divide in due parti: ricerca e valutazione statica. Si potrebbe assegnare tutto il tempo di elaborazione alla ricerca, lasciando alla valutazione statica solo il compito di riconoscere una

vittoria, un pareggio o una sconfitta, oppure si potrebbe fornire al giocatore tanta conoscenza da rendere superflua la ricerca: questi sono due casi limite difficilmente realizzabili, vuoi per il tempo che richiederebbe la prima, vuoi per la difficoltà (se non la impossibilità) di realizzare la seconda. È stato necessario quindi mediare le due situazioni, e limitare l' ampiezza della ricerca da una parte e la quantità di conoscenza dall'altra.
In definitiva, la funzione di valutazione statica che viene solitamente impiegata negli algoritmi di visita dell'albero di gioco scaturisce da considerazioni pratiche: una volta che l'esperienza ha portato alla determinazione della quantità di conoscenza da cui derivano le migliori prestazioni, si viene a definire un costo computazionale per la valutazione statica dei nodi, ed una profondità di ricerca limitata da esso.
È chiaro che se si riducesse ulteriormente questa quantità di conoscenza, la valutazione statica richiederebbe meno tempo, così da consentire di andare più in profondità nella ricerca, ma a questo punto, anche se si togliesse la porzione di conoscenza meno importante, potrebbe sempre verificarsi un caso in cui essa si dimostra indispensabile. L'idea alla base della distribuzione della conoscenza cerca di superare questo problema definendo più funzioni di valutazione ridotte, tali che la loro unione rappresenti l'intera conoscenza iniziale, ed associando ognuna di esse ad una istanza che esegue parallelamente alle altre l'algoritmo di visita dell'albero di gioco. Per ogni istanza ci sarà un guadagno di tempo che consentirà una ricerca più approfondita, e d'altra parte l'insieme di tutte le istanze avrà valutato tutta la conoscenza iniziale (anche se suddivisa in varie porzioni).
Al termine si avrà una decisione proposta da ogni istanza, e dall'insieme delle decisioni proposte sarà estratta quella definitiva. La perdita di completezza nella valutazione può essere così compensata dal guadagno di profondità nella ricerca.
Questo tipo di approccio comporta i seguenti problemi:

1. quante devono essere e quale conoscenza deve essere assegnata alle istanze (criterio di distribuzione). È chiaro che deve essere evitato che tutte le istanze prendano la stessa decisione, o che prendano tutte le decisioni ammissibili, perché in questi casi non si avrebbe nessun beneficio dal metodo. Ne consegue che la conoscenza delle varie istanze non deve essere troppo simile, per non rientrare nel primo caso; inoltre anche il numero di istanze deve essere adeguatamente proporzionato, perché poche istanze rischierebbero di rientrare nel primo caso, e troppe nel secondo.

2. come scegliere una tra le decisioni proposte dalle istanze (criterio di selezione). Esistono numerosi criteri possibili per selezionare una delle decisioni proposte; in [29], ad esempio, sono proposte le seguenti classi di criteri:

- a maggioranza generalizzata con pesi costanti: a ciascuna delle istanze viene associato un peso costante, dipendente dall'importanza della sua funzione di valutazione, e tale peso viene abbinato alla mossa che essa ha selezionato. Siccome una mossa può essere scelta anche da più istanze diverse, si definisce come peso della mossa la somma dei pesi che le sono stati abbinati dalle istanze che l'hanno scelta. Il criterio di selezione consiste nel selezionare la mossa di peso maggiore.
  (OSSERVAZIONE: quando il peso associato ad ogni istanza è unitario, il criterio è detto a maggioranza semplice);
- a maggioranza generalizzata con pesi variabili: è analogo al criterio precedente, però prevede che il peso associato ad ogni istanza non sia fisso, ma variabile a seconda della situazione contingente in cui deve essere eseguita la valutazione.
- visita selettiva a posteriori dell'albero di gioco: una volta ricavate le mosse proposte dalle varie istanze, si esegue una visita dell'albero di gioco considerando al primo livello solo tali mosse, ed utilizzando una funzione di valutazione completa.

2.5 Un dominio di applicazione: gli scacchi

L'analisi dei problemi scaturiti da questo nuovo approccio non può essere affrontata senza entrare nel merito del dominio di applicazione, in quanto in funzione del dominio di applicazione possiamo valutare nozioni come l'importanza di porzioni di conoscenza, o la determinazione di sottinsiemi di conoscenza da distribuire.
Il gioco degli scacchi rappresenta un ottimo banco di prova in questo senso, sia per le analisi che nel corso dei secoli ne hanno evidenziato le componenti strategiche più significative, e che possono quindi aiutare nell'esame della relativa conoscenza, sia per l'esistenza di metodi di confronto quali il sistema ELO (che permette di attribuire un valore assoluto alle qualità di un giocatore) o l'uso di test appositi (che permettono di valutare la bontà delle singole scelte di un giocatore), sia infine per la disponibilità di un giocatore artificiale di dominio pubblico già sperimentato (Gnuchess 4.0), con un doppio vantaggio: il suo codice può essere riusato per la creazione di nuovi giocatori, ed esso stesso può essere impiegato come metro di paragone per testare la forza dei nuovi giocatori creati.
Ci proponiamo dunque di realizzare e sperimentare un giocatore artificiale di scacchi che esegua la visita dell'albero di gioco secondo il metodo della distribuzione della conoscenza.
Esperimenti precedenti basati su questo metodo sono i seguenti:

- in [26] è stato estratto da un normale programma di scacchi un programma detto "esploratore", il quale esamina solamente le mosse tattiche, cioè

quelle mosse che mirano a conseguire un guadagno di materiale. Avendo da considerare solamente il materiale, un esploratore riesce a raggiungere profondità maggiori rispetto al programma originario. L'idea è quella di avere più programmi comunicanti: uno che esegue la normale ricerca posizionale, e gli altri che eseguono una ricerca più veloce fornendo un insieme di buone mosse tattiche. Per la selezione della mossa i programmi sono eseguiti parallelamente: al termine viene scelta la migliore mossa posizionale che rientra anche tra le mosse tattiche trovate dagli esploratori;

- in [3] è presentato lo studio di un giocatore basato su una distribuzione rudimentale della conoscenza: in esso ciascuna istanza di ricerca è costituita da un giocatore artificiale commerciale, ed al termine viene scelta la mossa che ha ottenuto il maggior numero di proposte;

- in [29] è stata esaminata la conoscenza della versione 3.1 del programma Gnuchess, e si è cercato di combinare più istanze di ricerca contenenti sottinsiemi della conoscenza di Gnuchess 3.1, mostrando con esperimenti che potenzialmente il giocatore parallelo risultante può essere più forte del giocatore sequenziale da cui ha origine. I metodi seguiti in questo lavoro saranno utilizzati più avanti per la definizione e sperimentazione del nostro giocatore parallelo.

2.6 Linda

Linda [10] è un modello di programmazione parallela consistente in alcune semplici operazioni che agiscono su un'area di memoria globale, condivisa tra i processi, detta spazio delle tuple. Le tuple sono gli oggetti che consentono la creazione e la coordinazione dei processi: sono definite come sequenze ordinate di campi ai quali possono essere associati o valori, o porzioni di codice. Nel secondo caso, l'esecuzione del codice ha inizio al momento della generazione della tupla, ed al termine dell'elaborazione si ha una sostituzione del codice stesso con il valore determinato.
Una tupla i cui campi contengono un valore è detta tupla-dato, ed è un oggetto passivo; una tupla in cui ci sono campi in corso di valutazione è detta tupla-processo, ed è un oggetto attivo. Quando tutti i campi di una tupla-processo sono stati valutati, essa diventa una tupla-dato.
Ogni processo ha accesso diretto ai dati contenuti nello spazio delle tuple per mezzo delle seguenti operazioni:

- out: viene depositata nello spazio delle tuple una nuova tupla-dato;

- eval: viene depositata nello spazio delle tuple una nuova tupla-processo (il sistema si preoccuperà di associare ai campi attivi della tupla altrettanti processori addetti alla loro valutazione);

- in: viene prelevata una tupla-dato dallo spazio delle tuple;
- rd: viene consultata una tupla-dato presente nello spazio delle tuple (in questo caso la tupla non viene rimossa, come invece avviene con l'operatore in).

Le operazioni in e rd sono bloccanti, cioè sospendono il processo che le invoca finché non viene trovata nello spazio delle tuple la tupla con le caratteristiche richieste (l'accesso alle tuple è di tipo associativo, e non per indirizzamento). In generale, le operazioni descritte hanno la proprietà di atomicità, così da garantire che più processi possano operare contemporaneamente sullo spazio delle tuple in maniera consistente.
La comunicazione tra i processi avviene attraverso messaggi depositati nello spazio delle tuple, e questo è consentito da due importanti proprietà del modello Linda di programmazione distribuita:

1. disaccoppiamento in spazio: un qualsiasi numero di processi con spazi degli indirizzi disgiunti può accedere ad una stessa tupla contenuta nello spazio delle tuple;
2. disaccoppiamento in tempo: una tupla inserita da un processo nello spazio delle tuple vi rimane anche dopo che il processo è terminato; in questo modo può essere letta anche da un altro processo che inizia dopo la terminazione del primo.

Lo spazio delle tuple si presenta così come un flessibile meccanismo che permette oltre alla normale comunicazione tra processi anche la comunicazione fra programmi scritti in linguaggi diversi, o fra un programma utente ed il sistema operativo, oppure ancora fra un programma ed una futura versione di se stesso.
La comunicazione tra processi e la loro creazione sono in pratica due aspetti della stessa operazione: si tratta in sostanza della generazione di una tupla, che nel primo caso sarà passiva (conterrà le informazioni da trasmettere ad altri processi) e nel secondo caso attiva (conterrà il codice del processo da creare). Visto che Linda permette anche di raggruppare le tuple in strutture, si possono così organizzare strutture dati distribuite (ogni elemento della struttura è un valore accessibile contemporaneamente da più processi) oppure collezioni di processi in strutture attive (ogni elemento della struttura è inizialmente un flusso di computazione indipendente dagli altri, ed al suo termine si trasforma in un dato; quando tutti i flussi sono terminati, si ha una struttura passiva che rappresenta il risultato della computazione complessiva).
In definitiva, la creazione esplicita dei processi consente un'estrema flessibilità nella costruzione dell'architettura logica del programma parallelo. Per quanto riguarda la coordinazione dei processi, Linda si rivela di straordinaria efficacia soprattutto nella gestione di moduli sequenziali predefiniti, nel

qual caso sono sufficienti solo pochi comandi di attivazione e sincronizzazione per ottenere una completa struttura parallela. In presenza infine di variabili condivise tra processi paralleli, grazie all'uso dello spazio delle tuple ed alla proprietà di atomicità degli operatori disponibili, viene realizzata una regolazione dell'accesso in mutua esclusione completamente trasparente al programmatore, con un vantaggio notevole nella prevenzione degli errori di programmazione.
Quanto finora descritto è il comportamento di un modello: non ha importanza come siano effettivamente realizzati i vari flussi di computazione, ma solo il modo in cui sono creati ed è loro permesso di cooperare. Il modello può essere messo in pratica in molti modi differenti ed in diversi contesti: esistono infatti realizzazioni per ambienti Lisp, Prolog, Pascal ed anche per ambienti object-oriented. Il supporto da usare per la realizzazione del giocatore artificiale di scacchi a conoscenza distribuita è il sistema Network C-Linda per reti di workstation [27]: si tratta di una particolare implementazione del modello Linda, destinata alla creazione e coordinazione di processi scritti in linguaggio C, ed indicata per un'architettura parallela come quella a nostra disposizione, cioè una rete locale di 11 SUN SparcStation collegate in Ethernet.
Sia la creazione dei processi che la loro interazione sono di solito rese disponibili direttamente dal sistema operativo, per cui si potrebbe anche evitare l'uso di un linguaggio di coordinamento. In realtà, ciò che rende preferibile l'utilizzazione di tale linguaggio è l'esistenza di un compilatore e di un ambiente di sviluppo e controllo dei programmi. Il compilatore è il componente più importante del sistema Network C-Linda; esso prevede l'esecuzione di due fasi distinte:

1. precompilazione del codice sorgente, con trasformazione delle operazioni Linda in sequenze ottimizzate di operazioni C, stabilite con particolare attenzione all'efficienza dell'esecuzione;

2. compilazione standard del codice C usato.

L'ambiente di programmazione Network C-Linda comprende anche un insieme di servizi volti ad assistere il programmatore nello sviluppo e nella correzione dei programmi: in particolare è fornito uno strumento (TupleScope) che consente una visualizzazione grafica degli oggetti che compongono un programma Network C-Linda, e ne descrive dinamicamente l'evoluzione durante l'esecuzione.
Dall'utilizzazione di Network C-Linda come linguaggio di coordinazione ci aspettiamo di realizzare un giocatore artificiale parallelo con un'architettura flessibile, in grado di gestire un numero di istanze non specificato a priori. Attraverso gli operatori propri del modello, è possibile inoltre controllare la coordinazione delle istanze inserendo solo poche linee di codice all'interno delle routine sequenziali tratte da Gnuchess 4.0, le quali potranno così essere

agevolmente riusate.

Capitolo 3

Gnuchess 4.0

Gnuchess 4.0 è un programma di scacchi di dominio pubblico (messo a disposizione dalla Free Software Foundation), scritto in linguaggio C, a cui molti programmatori hanno dedicato tempo e lavoro per migliorarne le prestazioni. Diverse contribuzioni sono arrivate e continuano ad arrivare da molte parti: ad esempio, sono state donate interfacce per X-windows e per SUN-tools, permettendo così una visualizzazione piacevole della scacchiera, ed inoltre sono state introdotte opzioni di compilazione che consentono di eseguire Gnuchess 4.0 anche su macchine con sistema operativo MSDOS, e su macchine con limitata capacità di memoria, come McIntosh e PC. Altre contribuzioni hanno riguardato l'aggiunta di grandi porzioni di libri delle aperture, e di raccolte di partite magistrali. In più, sono stati apportati miglioramenti nelle routine di visita dell'albero di gioco, sono state aggiunte euristiche ed a volte anche riscritte intere procedure. Nella sua evoluzione dalla versione iniziale, Il programma Gnuchess ha accresciuto la sua forza arrivando a possedere quella di un Maestro di scacchi, battendo altri programmi analoghi, come il Fidelity Mach 3 (classificato con il punteggio USCF 2265) e vincendo le ultime due manifestazioni del torneo annuale su piattaforma stabile fra giocatori artificiali di scacchi [7, 8].
Essendo un programma di gioco finalizzato alla massimizzazione delle prestazioni, e quindi alla minimizzazione dei costi computazionali, in esso è stata spesso sacrificata la modularità del codice. A questo si deve aggiungere che, forse a causa delle frequenti modifiche eseguite da numerosi programmatori, la documentazione del programma risulta spesso frammentaria ed insufficiente, e ciò rende particolarmente difficile la distinzione del flusso di controllo all'interno del codice.
La versione su cui abbiamo lavorato è la 4.0, patch level 62 (Settembre 1993): vedremo più avanti un compendio delle principali modifiche che la differenziano dalla versione 3.1 sulla quale è basato il lavoro di Tozzi [29].

3.1 Architettura generale

Gnuchess 4.0 è un sistema software che permette la disputa e l'analisi di partite di scacchi: esso è in grado di giocare contro se stesso, o contro un altro giocatore, rispettando tempi predefiniti e consentendo l'esecuzione di alcuni comandi volti alla gestione dell'input/output ed alla modifica delle modalità di gioco. In particolare (vedere figura 3.1), Gnuchess 4.0 realizza un ciclo all'interno del quale vengono eseguiti consecutivamente l'inserimento dei comandi o della mossa dell'utente (oppure la selezione della propria mossa nel caso che il programma stia giocando contro se stesso) e la selezione della mossa migliore da parte del calcolatore; al termine di entrambe le operazioni viene controllato il rispetto dei limiti di tempo e l'eventuale termine della partita (termine identificato sia da una situazione di stallo o di scacco matto, che dallo scadere del tempo concesso).
L'interruzione del ciclo avviene a causa del comando quit (o exit), mentre il termine di una partita lascia la possibilità di continuare il ciclo iniziandone un'altra.

3.2 Componenti di base

3.2.1 Rappresentazione dello stato del gioco

Lo stato del gioco è identificato dalla posizione dei pezzi e da informazioni generali, come il giocatore a cui tocca muovere, la possibilità di arroccare, ecc.
Una posizione è determinata dal contenuto delle caselle della scacchiera: per la sua rappresentazione sono utilizzati i due array board e color, di 64 elementi ciascuno, contenenti per ogni elemento l'informazione rispettivamente del tipo di pezzo e del colore del pezzo presente nella casella corrispondente. I tipi di pezzo sono pawn = pedone, knight = cavallo, bishop = alfiere, rook = torre, queen = regina e king = re, ed i colori sono white = bianco e black = nero; per indicare che una casella è vuota viene memorizzato nell'elemento corrispondente di entrambi gli array il valore NULL.
La corrispondenza tra gli elementi degli array e le caselle della scacchiera è definita in maniera biunivoca nel seguente modo. Una casella della scacchiera è identificata dalla riga e dalla colonna che occupa, quindi associando i numeri 0..7 alle righe da sinistra a destra e 0..7 alle colonne dal basso all'alto (prendendo come punto di vista quello del bianco), si identifica una casella con la coppia (r, c) dove $r, c \in [0..7]$; la funzione di codifica usata da Gnuchess 4.0 è

$$locn(r, c) = (r << 3) \mid c \quad \forall r, c \in [0..7]$$

mentre la funzione di decodifica è suddivisa nelle due funzioni

$$row(sq) = sq >> 3 \quad \forall sq \in [0..63]$$

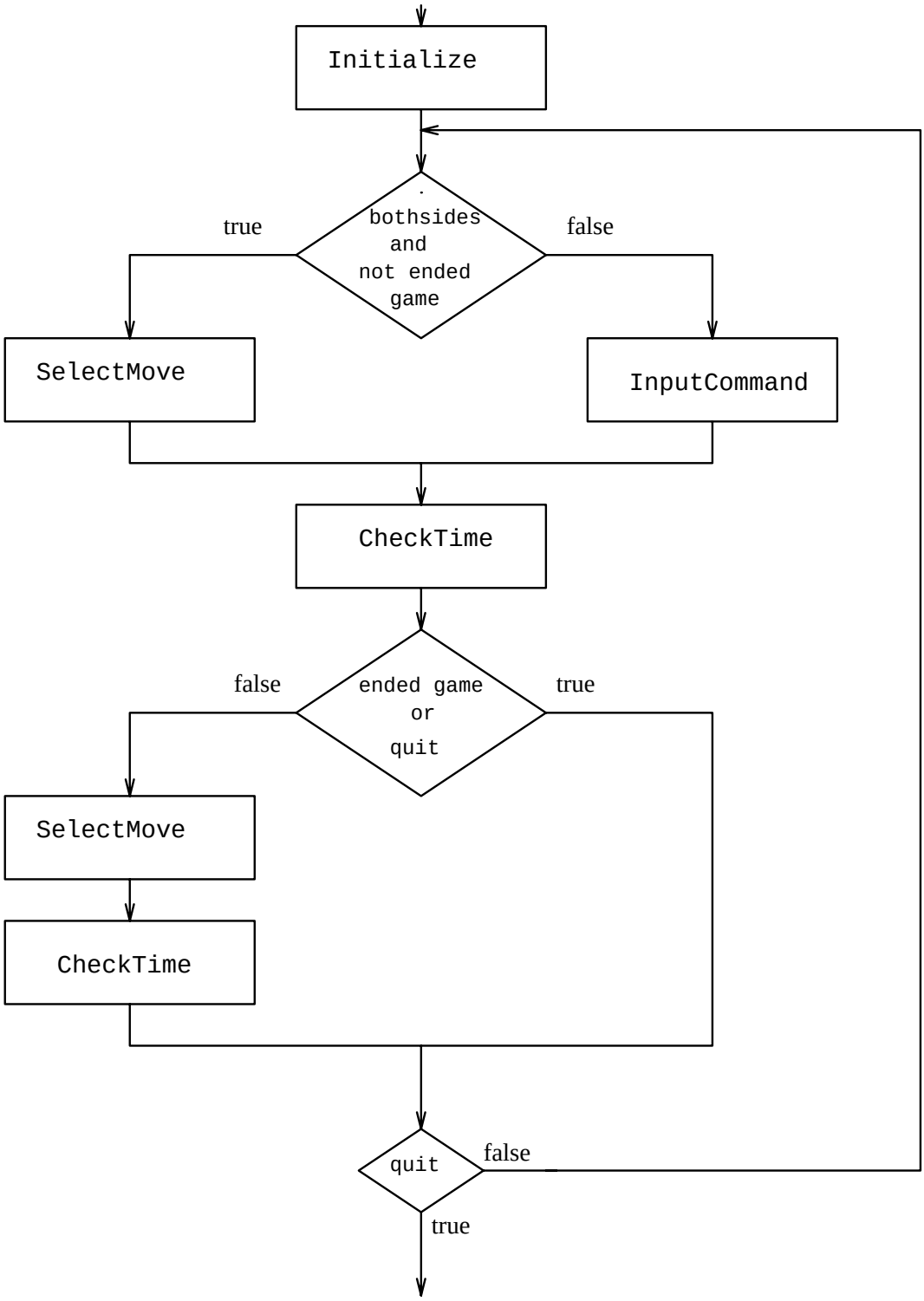

Figura 3.1: Ciclo di base di Gnuchess 4.0

$$column(sq) = sq \ \& \ 7 \quad \forall sq \in [0..63]$$

In aggiunta è previsto anche un array di comodo (PieceList) che contiene per ogni giocatore la sequenza delle caselle occupate dai suoi pezzi (non è previsto nessun ordinamento all'interno di PieceList, se non il fatto che il primo elemento indica la posizione del re); quest'ultimo array non è di per sé sufficiente ad identificare univocamente la posizione, ma è utile per velocizzare alcune procedure a cui serve sapere in quali caselle si trovino i pezzi.
Alcune tra le variabili che concorrono a rappresentare lo stato di gioco sono:

- player: è il giocatore a cui tocca muovere;
- castld[side]: indica per ogni giocatore se ha già eseguito l'arrocco;
- Mvboard[side]: indica per ogni casella della scacchiera quante volte un pezzo presente su di essa è stato mosso (serve tra l'altro per determinare se re o torre sono già stati mossi, nel qual caso non è più ammesso l'arrocco);
- epsquare: indica in quale casella il giocatore che deve muovere può eseguire una cattura en passant (-1 se non è possibile);
- GameCnt: contiene il numero di semimosse che sono state eseguite dall'inizio della partita;
- Game50: contiene il numero d'ordine dell'ultima mossa di pedone o cattura di pezzo (serve per controllare il rispetto della regola delle 50 mosse, secondo la quale una partita è considerata pari se per 50 mosse non sono state eseguite né mosse di pedone, né catture di pezzi; se GameCnt $-$ Game50 > 99, allora la partita è pari).

3.2.2 Rappresentazione dell'albero di gioco

Siccome non è conveniente mantenere una rappresentazione statica dell'albero di gioco, Gnuchess 4.0 utilizza alcune strutture di dati che consentono di rappresentarne solamente la porzione che interessa momento per momento.
A partire dalla posizione da valutare, posta alla radice dell'albero, vengono sviluppati durante la visita dell'albero soltanto i dati riguardanti il cammino che porta alla posizione attualmente in esame: per ogni nodo (posizione) del cammino sono indicati tutti i possibili archi uscenti (le mosse), e sarà poi la procedura di visita dell'albero che si preoccuperà di tenere traccia di quali archi sono stati effettivamente attraversati per arrivare al nodo terminale del cammino (posizione attualmente in esame).
La struttura atta a contenere i dati di una mossa è la seguente:

```
struct leaf
{
   short int f;
   short int t;
   short int  score;
   short int  reply;
   short int  width;
   unsigned short flags;
};
```

In essa sono contenute le informazioni riguardanti

- la casella di partenza del pezzo che muove (f);
- la casella di arrivo del pezzo che muove (t);
- il punteggio assegnato alla mossa (score);
- il numero di mosse previste per il turno successivo (width);
- alcuni dati statistici relativi alla mossa (flags), e più in particolare:
  - book: la mossa è stata scelta dal libro delle aperture;
  - draw: la mossa determina una situazione di patta;
  - capture: la mossa è stata una cattura di un pezzo;
  - check: la mossa è stata uno scacco al re;
  - pwnthrt: la mossa minaccia la promozione di un pedone;
  - exact: la valutazione della mossa deriva da un valore esatto, cioè dal valore di un nodo terminale dell'albero di gioco, e non dalla valutazione statica di un nodo intermedio;
  - epmask: la mossa è una cattura en passant;
  - cstlmask: la mossa è un arrocco;
  - promote: la mossa è la promozione di un pedone;
  - pmask: se la mossa è la promozione di un pedone, qui c'è la codifica del pezzo promosso.

Per la rappresentazione dell'intero cammino sono usate le due variabili globali

```
struct leaf Tree[TREE];
short TrPnt[MAXDEPTH];
```

Tree contiene in sequenza tutte le mosse, nodo dopo nodo; TrPnt contiene ordinatamente l'indicazione dei punti di Tree in cui inizia la serie di mosse associate ad un particolare nodo.
In definitiva, indicando con ply il livello di profondità dei nodi rispetto alla

radice, si ha che se in Tree[x], Tree[x+1], ... , Tree[x+n] ci sono le n mosse possibili al ply-esimo livello, allora sarà TrPnt[ply]=x e TrPnt[ply+1]=x+n+1 (oppure TrPnt[ply+1]=x se ply è il livello corrispondente al nodo terminale del cammino).

3.2.3 Generazione delle mosse

L'algoritmo di generazione delle mosse di Gnuchess 4.0 è derivato da uno studio di H. E. Sandstroem per la creazione di un generatore di mosse hardware, ed è estremamente efficiente. L'idea generale è quella di precalcolare ogni possibile mossa che ogni pezzo può eseguire partendo da ogni casella, indipendentemente dalla dislocazione degli altri pezzi sulla scacchiera. Questi dati precalcolati sono memorizzati nei due array nextpos[piece][from][to] e nextdir[piece][from][to], dai quali verranno prese le informazioni al momento opportuno.
L'array nextpos contiene l'indicazione della casella successiva che piece può raggiungere lungo la direzione che l'ha portato da from a to; l'array nextdir contiene invece l'indicazione della prima casella in cui piece può andare partendo da from e seguendo una direzione diversa da quella che conduce a to. Grazie all'uso di queste strutture precalcolate, l'unica cosa da fare quando si generano le mosse è controllare le collisioni tra i pezzi e scegliere l'array opportuno in cui andare a prelevare i dati.
Ad esempio, la prima mossa che può fare una regina nella casella e8 è memorizzata in nextpos[queen][e8][e8]. Supponiamo che la prima casella di destinazione sia e7, e che la casella e7 sia occupata: in tal caso la prossima mossa sarà trovata in nextdir[queen][e8][e7]; se invece la casella e7 fosse stata libera, allora la prossima mossa sarebbe stata trovata in nextpos[queen][e8][e7].
Si noti però che non viene richiesto il colore del pezzo che deve muovere, e questo provoca un problema quando deve essere esaminato un pedone. Il pedone, infatti, è l'unico pezzo che muove diversamente a seconda del colore che ha, dovendo sempre avanzare verso il lato opposto della scacchiera. Per questo motivo è stato definito un tipo di pezzo in più (bpawn = pedone nero), al quale corrispondono negli array nextpos e nextdir specifiche mappe dei movimenti.
Un altro problema che sorge dall'applicazione di questo metodo di generazione delle mosse è che vengono generate anche quelle mosse che lasciano il re sotto scacco (mosse pseudolegali): tali mosse vengono riconosciute ed eliminate in fasi successive assegnando loro punteggi talmente bassi da far sì che non possano mai essere selezionate.
La procedura che determina il contenuto degli array nextpos e nextdir è Initialize_moves(), ed è eseguita una sola volta all'avvio di Gnuchess 4.0.
Le mosse vengono generate man mano che la procedura di visita va in profondità nell'albero di gioco. La procedura che genera tutte le mosse possibili (legali e pseudolegali) al ply-esimo livello di profondità è MoveList(ply), la

quale per ogni pezzo sulla scacchiera (individuabile in maniera efficiente con l'array PieceList) determina tramite nextpos e nextdir tutte le possibili caselle in cui il pezzo può arrivare, e memorizza le coppie <casa di partenza, casa di arrivo>, rappresentanti le mosse, nella posizione opportuna della struttura che rappresenta l'albero di gioco.
Esiste anche un'altra procedura analoga di generazione delle mosse, CaptureList(ply), la quale viene utilizzata per la ricerca quiescente in quanto non considera quelle mosse che non eseguono la cattura di un pezzo avversario.

3.2.4 Dati della partita in corso

Tutti i dati rilevanti relativi ad ogni mossa vengono memorizzati in una struttura così composta:

```
struct GameRec
{
  unsigned short gmove;
  short score;
  short depth;
  long time;
  short piece;
  short color;
  short flags;
  short Game50;
  long nodes;
  unsigned long hashkey, hashbd;
  short epssq;
};
```

I campi di GameRec hanno il seguente significato:

- gmove: è la codifica della mossa eseguita. Dato che la mossa è rappresentata dalla coppia $from$ = casella di partenza e to = casella di arrivo, con $from, to \in [0..63]$ come indicato nella rappresentazione della posizione, allora la sua codifica è ottenuta ponendo il valore di $from$ negli 8 bit più significativi di un unsigned short, ed il valore di to negli 8 bit meno significativi:

  $$movecod = (from << 8) \mid to$$

  La decodifica sarà ovviamente ottenuta estraendo dalla mossa codificata i valori dei bit opportuni:

  $$from = movecod >> 8$$

  $$to = movecod \ \& \ (OOFF)_{16}$$

- score: è il valore assegnato alla mossa dal giocatore che l'ha eseguita;
- depth: è la profondità di ricerca che é stata raggiunta nella valutazione dell'albero di gioco;
- time: è il tempo impiegato dal giocatore che ha eseguito la mossa per effettuare la sua scelta;
- piece: è il tipo del pezzo eventualmente catturato;
- color: è il colore del pezzo eventualmente catturato;
- flags: vengono indicate particolari condizioni determinate dalla mossa eseguita, le stesse condizioni previste per la rappresentazione dell'albero di gioco;
- Game50: contiene il numero d'ordine dell'ultima mossa di pedone, o cattura di un pezzo;
- nodes: è il numero di nodi dell'albero di gioco che sono stati visitati per la scelta della mossa eseguita;
- hashkey, hashbd: contengono rispettivamente l'indirizzo della tabella delle trasposizioni in cui può essere memorizzata la valutazione della posizione esaminata, e l'identificatore della posizione esaminata, da usare per riconoscere le collisioni.
- epssq: è la casella in cui può essere eseguita la cattura en passant dopo l'esecuzione di questa mossa (ha valore -1 se non è possibile eseguire l' e.p.).

Esiste un array globale GameList, composto da elementi di tipo GameRec, in cui viene memorizzata la sequenza di mosse che porta alla posizione in esame. È da questa variabile che vengono prelevate le informazioni da salvare su un archivio permanente al termine dell'esecuzione di Gnuchess 4.0 o a seguito dei comandi list o save, ed è a questa variabile che fa riferimento la procedura di selezione della mossa per mantenere le informazioni statistiche necessarie durante la visita ricorsiva dell'albero di gioco.

3.3 Il controllo del tempo

Gnuchess 4.0 consente due diverse modalità di controllo del tempo:

1. tempo per mossa: non vi è controllo nel tempo globale della partita, ma solo nel tempo di generazione della mossa da parte del giocatore artificiale;

2. uso dell'orologio: devono essere eseguite un certo numero di mosse in un determinato ammontare di tempo. È possibile eseguire fino a 4 controlli diversi all'interno della stessa partita (ad esempio, le prime 40 mosse in 5 minuti, poi le successive 20 in 10 minuti, ecc.). In questo caso viene utilizzata per il controllo del tempo una variabile così definita:

```
struct TimeControlRec
{
  short moves[2];
  long  clock[2];
};

struct TimeControlRec TimeControl;
```

   Dal numero di mosse che mancano prima del controllo del tempo, contenuto per ogni giocatore *side* in TimeControl.moves[*side*], e dal tempo rimanente, contenuto per ogni giocatore *side* in TimeControl.clock[*side*], il giocatore artificiale ricava ogni volta il tempo da dedicare alla mossa successiva. Quando TimeControl.clock[*side*] diventa nullo e dal contenuto di TimeControl.moves[*side*] risulta che ci sono ancora mosse da fare, *side* ha perso la partita; se diventa nullo TimeControl.moves[*side*], invece, mentre dal contenuto di TimeControl.clock[*side*] risulta che c'è ancora tempo, vengono aggiornati entrambi i campi, aggiungendo a moves[*side*] il numero di mosse consentite prima del prossimo controllo, e a clock[*side*] il tempo concesso per queste mosse.

A seconda della modalità di controllo del tempo scelta, memorizzata nella variabile booleana TCflag (false ⇒ tempo per mossa ; true ⇒ uso dell'orologio), Gnuchess 4.0 calcola il tempo base da dedicare alla visita dell'albero di gioco (tempo che viene memorizzato nella variabile ResponseTime), e determina inoltre un tempo straordinario, da aggiungere al tempo base in quelle particolari condizioni che rendono necessario un approfondimento ulteriore della ricerca (tempo che viene memorizzato nella variabile ExtraTime);infine inizia la ricerca. Per sapere quando scade il tempo consentito, e potere così interrompere la ricerca, Gnuchess 4.0 esegue un controllo del tempo ogni ZNODES nodi, e setta il segnale di interruzione (flag.timeout) se al momento del controllo il tempo trascorso dall'inizio della ricerca (et) supera il tempo consentito per quella mossa (ResponseTime + ExtraTime).

3.4 La scelta della mossa

La procedura di selezione della mossa da parte di Gnuchess 4.0 (SelectMove) prevede l'esecuzione, secondo le modalità dell'Iterative Deepening, di un algo-

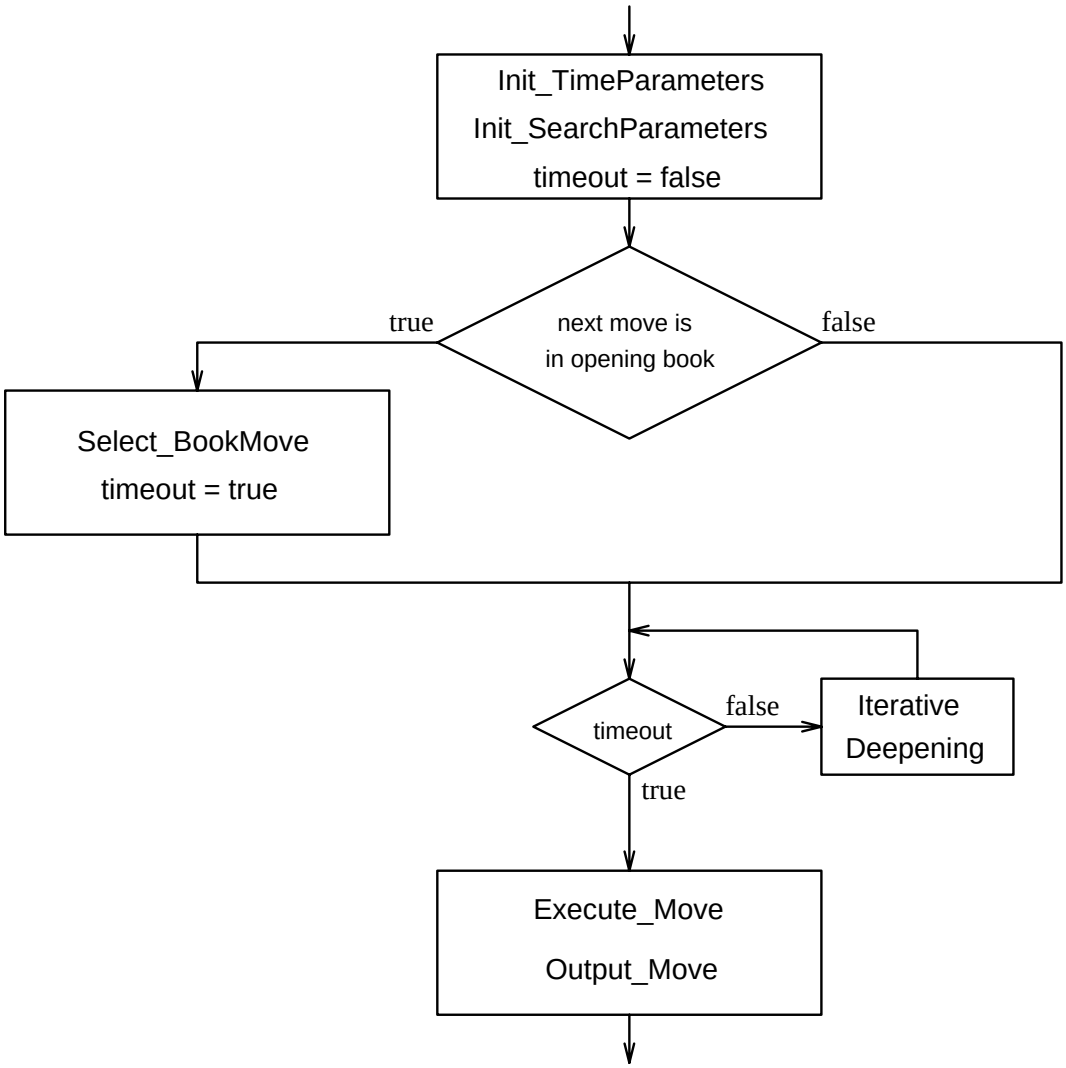

Figura 3.2: SelectMove

ritmo di tipo Aspiration Search, versione Fail Soft, come definito nel capitolo precedente. Innanzitutto (vedere figura 3.2) vengono determinati il tempo consentito per la risposta, calcolato a seconda della modalità prescelta di controllo del tempo, e la dimensione dei parametri di ricerca (in particolare, la finestra $\alpha\beta$ iniziale è definita rispetto al valore statico della posizione attuale).
Successivamente, a meno che non possa essere trovata una risposta nel libro delle aperture, viene eseguito un ciclo in cui la visita dell'albero è ripetuta più volte, aumentando ogni volta la profondità di ricerca, fintanto che non scade il tempo a disposizione. Al termine, la mossa selezionata viene eseguita e comunicata all'avversario.

Ad ogni iterazione, viene eseguito l'algoritmo Aspiration Search in combinazione con l'AlphaBeta in versione NegaMax/FailSoft. Si deve però riscontrare la seguente anomalia: le opportunità fornite dal valore ritornato in caso di fal-

limento iniziale della ricerca non sono utilizzate, in quanto nella eventuale visita successiva viene sempre usata la finestra $\alpha\beta$ canonica $]-\infty, +\infty[$, diversamente da quanto previsto sia per AspirationSearch che per F_AspirationSearch.
Per la valutazione statica dei nodi terminali dell'albero ridotto, Gnuchess 4.0 si serve della procedura *evaluate*(), che esamineremo accuratamente più avanti. In alternativa è previsto l'uso della tabella delle trasposizioni, consistente in una tabella di dimensioni variabili tra un minimo di 8001 byte ed un massimo ricavato al momento dell'esecuzione del programma dalla quantità di memoria libera disponibile (se la memoria libera ha dimensione minore di 8001 byte, la tabella delle trasposizioni non può essere utilizzata). Per poter gestire le partite che Gnuchess 4.0 gioca contro se stesso, sono create due tabelle delle trasposizioni, una per ogni colore: esse sono contenute in due aree di memoria distinte, a cui si può accedere tramite il puntatore ttable[side].
In ogni elemento della tabella sono contenuti l'identificatore della posizione (hashbd), la profondità raggiunta nella valutazione della posizione (depth), la codifica della mossa migliore da giocare (mv), alcuni dati relativi a tale mossa (flags) ed il punteggio assegnatole (score); le condizioni che consentono l'uso della tabella delle trasposizioni al posto della valutazione per mezzo di *evaluate*() dipendono dal confronto della profondità di valutazione raggiunta per il punteggio contenuto nella tabella con la profondità di attuale valutazione dell'albero di gioco.
In aggiunta, è possibile utilizzare anche una versione permanente della tabella delle trasposizioni, l'hash file: vedremo più avanti le sue caratteristiche.

3.4.1 Visita dell'albero di gioco

L'algoritmo F_AlphaBeta_NgMx esegue la visita dell'albero di gioco utilizzando dinamicamente le variabili globali che concorrono a rappresentare lo stato del gioco e i dati della partita. Ogni volta che viene esplorata una nuova mossa, questa è effettivamente eseguita per mezzo della funzione MakeMove(), così che lo stato di gioco risultante diventa lo stato di gioco attuale, e nell'array che rappresenta la partita (GameList) viene aggiunta la mossa eseguita con tutti i relativi dati statistici. Ogni volta invece che la ricorsione fa tornare indietro di una mossa per consentire l'esame di una sua alternativa, la mossa è disfatta per mezzo della funzione UnmakeMove(), in modo da ripristinare la situazione precedente di stato di gioco e dati della partita.
Le mosse da esaminare sono generate ogni volta che si giunge ad una nuova posizione, e sono inserite nell'array Tree rispettando un ordinamento che tende ad ottimizzare l'efficienza dell'algoritmo AlphaBeta. In particolare, viene posizionata in modo da essere esaminata per prima la mossa che era risultata migliore al termine dell'iterazione precedente; subito dopo vengono le mosse che catturano l'ultimo pezzo mosso; seguono le altre mosse di cattura, in ordine di importanza del pezzo catturato; sono considerate infine le

mosse killer (memorizzate durante l'iterazione precedente negli array killr0, killr1, killr2 e killr3 a seconda della profondità dell'albero di gioco in cui sono state determinate) seguite dalle altre mosse. Durante la valutazione di una posizione, un puntatore provvede a scandire le mosse generate per essa, dalla prima all'ultima (a meno di eventuali tagli), seguendo l'ordinamento definito durante l'inserimento nell'albero di gioco.
Esiste un array globale PrVar[MAXDEPTH] in cui è memorizzata, durante la visita, la sequenza di mosse (codificate analogamente alle mosse della partita) che ha ottenuto fino a quel momento la migliore valutazione; al termine della visita, PrVar conterrà la variante principale, cioè la sequenza di mosse che Gnuchess 4.0 considera migliore per entrambi i giocatori.
La mossa selezionata per ogni iterazione è quella che ha ottenuto il maggiore punteggio, o a parità di punteggio è quella che prevede di avere il maggior numero di possibilità al turno successivo; essa è individuata tra le mosse del livello 0 dell'albero dal puntatore root, ed il punteggio assegnatole servirà per la determinazione della finestra $\alpha\beta$ iniziale dell'iterazione successiva.
La profondità di ricerca stabilita per ogni iterazione è subordinata alla ricerca quiescente, e può così essere superata nella valutazione delle posizioni instabili (quelle cioè che derivano da catture di pezzi, da scacchi al re e da particolari minacce).

3.4.2 Libro delle aperture

Gnuchess 4.0 può usare un libro delle aperture contenente informazioni codificate (binbook file), ed anche un libro aggiuntivo contenente informazioni in formato ASCII (book file).
Al momento dello startup del programma, Gnuchess 4.0 carica il binbook file in memoria, dopodiché controlla se esiste il book file: se esiste, aggiunge il suo contenuto ai dati in memoria, ottenendo così un libro composto dall'unione di binbook file e book file, poi salva il tutto all'interno dello stesso binbook file, ampliando in questo modo il libro delle aperture disponibile. In ogni caso, la dimensione complessiva del libro delle aperture non deve superare il numero di mosse contenuto nella costante BOOKSIZE (attualmente risulta essere BOOKSIZE=250000).
Il book file da aggiungere deve sottostare ad una precisa sintassi: il contenuto richiesto consiste in una sequenza di aperture composte ognuna da una intestazione, prefissata dal carattere !, e seguita da una serie di mosse consecutive in notazione algebrica, separate tra loro da uno spazio bianco. Vediamo un esempio del formato che deve avere un bookfile:

```
! Four Knight's Game
e2e4  e7e5
g1f3  b8c6
```

```
b1c3  g8f6
f1b5  f8b4
o-o   o-o
d2d3  b4c3
b2c3  d7d6
c1g5  d8e7
f1e1  c6d8
d3d4  d8e6
! Giuoco Piano
e2e4  e7e5
g1f3  b8c6
f1c4  f8c5
d2d3  g8f6
b1c3  d7d6
c1g5  h7h6
g5f6  d8f6
c3d5  f6d8
```

Nel caricamento in memoria del libro delle aperture, viene creata una lista di linee di gioco, in cui ogni elemento è identificato da un puntatore ad un'area di memoria contenente le mosse relative. Al momento della ricerca nel libro delle aperture, vengono esaminate tutte le linee di gioco presenti, alla ricerca di quelle che corrispondono alla linea di gioco effettivamente giocata: se ne viene trovata almeno una, allora Gnuchess 4.0 sceglie casualmente una delle possibili continuazioni previste, altrimenti deve provvedere autonomamente alla determinazione della prossima mossa effettuando la valutazione dell'albero di gioco.
Le mosse sono memorizzate insieme all'indicazione della posizione risultante, per cui sono riconosciute anche trasposizioni rispetto alla linea di gioco effettivamente giocata: la costante BOOKFAIL rappresenta il numero di fallimenti consentiti prima dell'abbandono della ricerca nel libro delle aperture.

3.4.3 Hash file

Come ampliamento dell'euristica della tabella delle trasposizioni, che evita di valutare le stesse posizioni che si presentano nell'ambito della stessa partita, Gnuchess 4.0 ha la possibilità di memorizzare valutazioni ottenute in partite precedenti, per mezzo di un apposito archivio detto hash file.
L'hash file deve essere inizialmente creato ed opportunamente dimensionato attraverso una apposita opzione del comando di invocazione di Gnuchess 4.0 (la dimensione può variare fino ad un massimo di 2^{23} bytes, e deve comunque essere una potenza di 2), dopodiché Gnuchess 4.0 accede ad esso durante il gioco, sia in lettura durante la ricerca nell'albero , sia in scrittura quando ha ottenuto valutazioni di una certa affidabilità.

L'uso dell' hash file, essendo l'accesso ad un archivio permanente una operazione costosa, è limitato dai due parametri HashDepth e HashMoveLimit. Infatti, le valutazioni possono essere memorizzate in esso solo se sono state eseguite ad una profondità pari almeno ad HashDepth, e ad almeno HashMoveLimt mosse dall'inizio della partita (i due parametri assumono di default rispettivamente i valori 4 e 40, ma possono essere cambiati dall'utente per mezzo del comando hashdepth durante lo svolgimento di una partita).
L'esperienza ha dimostrato che l'utilizzo dell'hash file non peggiora le prestazioni di Gnuchess 4.0, ma non è chiaro se esso procuri un reale vantaggio.

3.5 L'interprete dei comandi

Gnuchess 4.0 fornisce all'utente la possibilità di inserire comandi di diverso tipo, oltre alla comunicazione della mossa eseguita, attraverso la procedura InputCommand().
I comandi previsti consentono di intervenire su:

- modifica dei parametri di ricerca;
- visualizzazione;
- modifica dei flag;
- controllo delle variabili;
- input/output;
- gestione della partita;

Se l'input non rappresenta uno dei comandi ammessi (di cui si può trovare una rassegna nell'appendice A), allora è interpretato come la comunicazione della mossa successiva: in tal caso viene realizzata una verifica di legalità. Se la verifica risulta positiva, allora la mossa viene eseguita, altrimenti viene visualizzato un messaggio di errore, in quanto l'input non è stato riconosciuto. Nella figura 3.3 si può vedere il controllo del flusso della procedura che interpreta i comandi, con particolare riferimento ai comandi quit e both, ed al controllo ed esecuzione della mossa comunicata dall'utente.

3.6 La funzione di valutazione

Dall'esame del codice di Gnuchess 4.0 si sono ricavate 8 categorie di conoscenza, di cui diamo una descrizione esemplificativa.

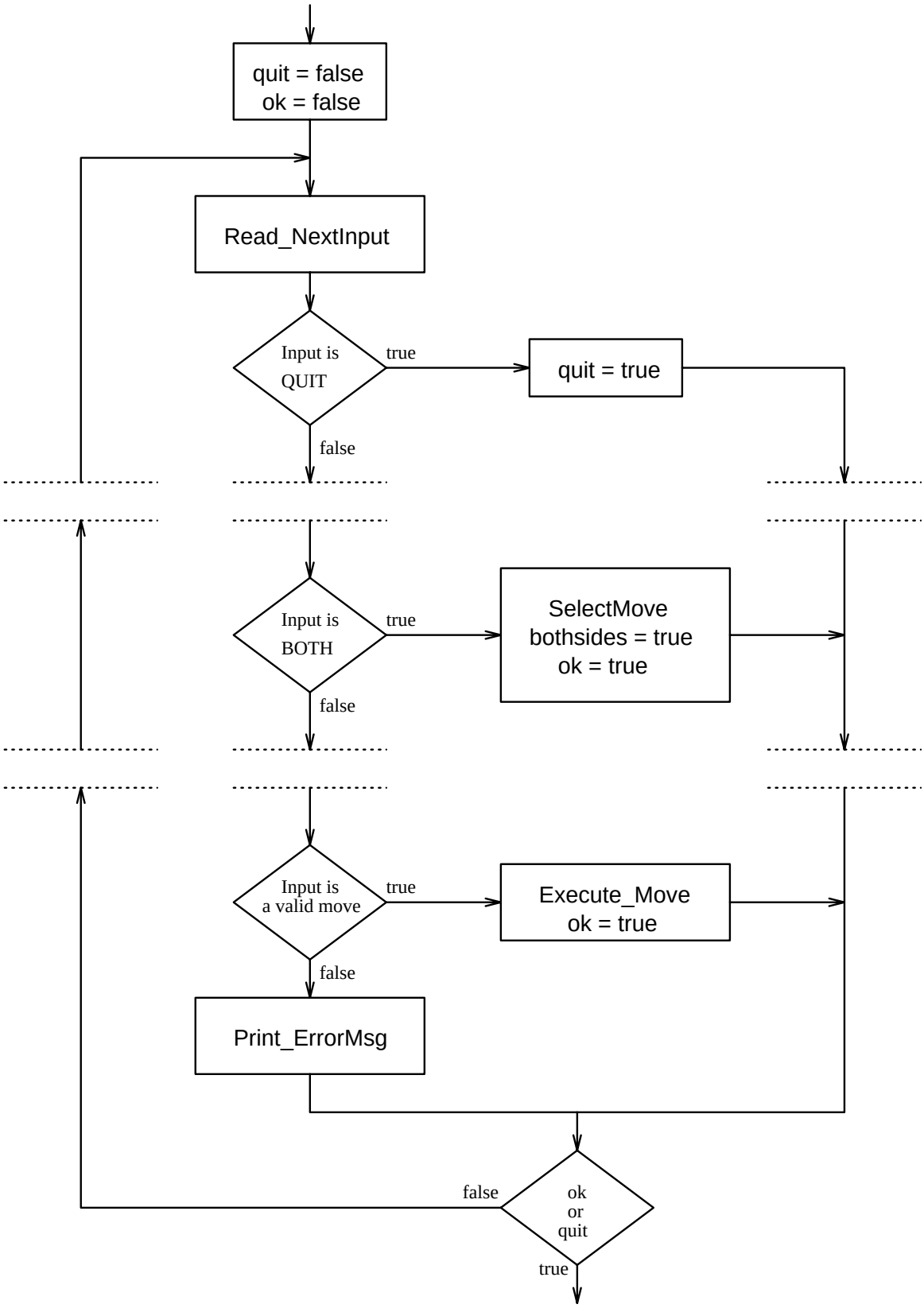

Figura 3.3: InputCommand

3.6.1 Euristiche

1. MATERIALE (m): Chiunque sappia giocare a scacchi conosce la differenza di valore esistente tra i pezzi. Una delle prime cose che si insegnano ai principianti, infatti, è che, a parte il re, la regina è il pezzo più importante (grazie al fatto che ha più possibilità di movimento), oppure che un alfiere ha press'a poco lo stesso valore di un cavallo, ecc. Tali valutazioni derivano dall'esperienza di gioco, e costituiscono la base indispensabile per qualsiasi giocatore di scacchi. Questa euristica si propone quindi di valutare le forze in campo in senso astratto, senza cioè considerare la loro disposizione sulla scacchiera.

2. BUONA SISTEMAZIONE DEI PEZZI (b): È sempre da tenere in considerazione la sicurezza dei propri pezzi rispetto ai possibili attacchi dell'avversario: devono infatti essere premiati i pezzi che non possono essere scacciati da dove si trovano, e penalizzati invece i pezzi che si trovano in caselle minacciate in maniera considerevole dai pezzi nemici.

3. SPAZIO E MOBILITÀ (x): Questa euristica consiste nel valutare la libertà di movimento che hanno i pezzi in gioco relativamente alla posizione che occupano. È chiaro infatti che si debba preferire una posizione in cui i propri pezzi abbiano un ampio raggio di azione, piuttosto che una posizione in cui essi siano costretti all'inattività per mancanza di spazio, o siano addirittura bloccati.

4. SICUREZZA DEL RE (k): Un altro aspetto importante da tenere in considerazione è quello della salvaguardia del proprio re. È assolutamente inutile infatti avere un gran vantaggio di pezzi o di spazio quando il proprio re è in pericolo e rischia di soccombere da un momento all'altro.

5. CONTROLLO DEL CENTRO (c): Secoli di pratica scacchistica hanno ormai reso indiscutibile l'importanza rivestita dal centro della scacchiera nell'analisi strategica di una partita, soprattutto nella fase iniziale o nel mediogioco, quando sono ancora presenti molti pezzi e non si ha un obiettivo più concreto da perseguire immediatamente.

6. STRUTTURA PEDONALE (p): I pedoni sono elementi spesso decisivi, sia positivamente che negativamente, perché, essendo i pezzi di valore minore, possono creare delle catene inattaccabili quando si proteggono l'un l'altro, oppure, essendo i pezzi con minore capacità di movimento, possono essere facilmente bloccati e catturati dai pezzi avversari. Una migliore struttura pedonale inoltre è una garanzia di superiorità nel finale della partita, quando si è ridotto il numero dei pezzi sulla scacchiera ed il pedone assume una maggiore importanza relativa.

7. POSSIBILITÀ DI ATTACCO (a): Un'altra euristica è quella che valuta la posizione in funzione di eventuali piani di azione, dettati da considerazioni strategiche legate al raggiungimento di obiettivi concreti, come lo sfruttamento delle debolezze avversarie, la promozione di un pedone, o l'attacco diretto al re avversario.

8. RELAZIONE TRA I PEZZI (r): È importante infine considerare la coesione che hanno i pezzi di uno stesso colore: esistono infatti combinazioni di pezzi che costituiscono di per se stesse un vantaggio (l'esempio più noto è la coppia degli alfieri), mentre in certe fasi della partita può portare giovamento la presenza di un tipo di pezzo piuttosto che un altro.

Chiaramente, queste euristiche non sono affatto indipendenti le une dalle altre. Una semplice cattura eseguita da un pedone, ad esempio, può coinvolgere contemporaneamente le euristiche b (viene minacciato un pezzo avversario), p (modifica della struttura dei pedoni), k (indebolimento delle difese del re), a (attacco contro il re avversario), c (controllo di una casella del centro), x (modifica di spazio e mobilità per entrambi i colori).

3.6.2 Ripartizione delle euristiche

Un programma di scacchi ha due parti distinte: la parte che esegue ed analizza le mosse, e la conoscenza che permette al programma di giocare bene. Generalmente, la prima è ben formalizzata e completa, mentre la seconda consiste di frammenti di programma designati a riconoscere importanti caratteristiche delle posizioni e, per ogni caratteristica trovata, a definire un peso che ne rifletta la qualità. I pesi saranno poi combinati linearmente per formare una valutazione numerica totale della posizione.
La procedura di Gnuchess 4.0 che determina la valutazione statica di una posizione è $evaluate(side, ply, \alpha, \beta, INCscore)$; essa calcola il valore statico della posizione attuale nel seguente modo.
Nell'array Pscore[ply-1] si trova il valore posizionale complessivo (senza considerare il materiale) determinato alla mossa precedente, mentre nella variabile INCscore si trova il contributo al valore posizionale che era derivato da un pezzo eventualmente mangiato nell'ultima mossa eseguita (e quindi non più presente nella scacchiera), oltre ad un valore addizionale determinato da eventuali modifiche alla struttura pedonale causate dall'ultima mossa (si fa notare che quest'ultimo valore fa parte di una porzione di conoscenza relativa all'euristica p); in definitiva, con Pscore[ply-1]+INCscore si ottiene una stima del valore statico della posizione attuale (senza considerare il materiale) determinata dal punto di vista di otherside. Valutando dal punto di vista di side e considerando anche il materiale si ricava la stima del valore posizionale attuale s per mezzo della formula $s = -Pscore[ply-1] - INCscore + \Delta mtl$. Se la stima s è compresa nell'intervallo $[\alpha - xwndw, \beta + xwndw]$, dove $xwndw$

è un ampliamento della finestra $\alpha\beta$ con valore 30 di default, ma con possibilità di essere modificato per mezzo del comando xwndw, viene eseguita una valutazione statica effettiva per mezzo della procedura $ScorePosition(side)$, altrimenti si tiene per buono il valore stimato s.
Nel caso particolare che uno dei due giocatori abbia solamente il re, e l'altro non abbia pedoni oppure non abbia pezzi, allora viene utilizzato un tipo di conoscenza ad hoc che riguarda la valutazione di facili finali di partita: ad esempio Gnuchess 4.0 possiede routine apposite per la valutazione statica di finali di re contro re e pedoni (con una conoscenza particolare quando c'è solo un pedone) e di re contro re più alfiere e cavallo.

La valutazione effettiva eseguita da $ScorePosition(side)$ consiste in:

- generazione di un punteggio $static\_score(p, side)$ per ogni pezzo p presente sulla scacchiera (considerando sempre il punto di vista di side);
- somma successiva dei punteggi dei pezzi dello stesso colore con la eventuale aggiunta ad ogni colore di un valore addizionale dipendente dalla euristica b: per ogni colore x si ha così

$$pieces\_val[x] = \sum_{\forall pezzo\ p\ di\ x} static\_score(p, side) + extra(x)$$

- differenza dei valori così ottenuti:

$$ScorePosition(side) = pieces\_val[side] - pieces\_val[otherside]$$

Entriamo nel dettaglio delle funzioni $static\_score(p, side)$ e $extra(x)$, evidenziando la suddivisione tra le varie euristiche presentate precedentemente.
La funzione $static\_score(p, side)$ ritorna un valore dipendente oltre che dal tipo del pezzo p e dal giocatore $side$ rispetto a cui si esegue la valutazione, anche dalla casella della scacchiera in cui il pezzo è posizionato e dallo stadio della partita in cui ci si trova: vediamo pezzo per pezzo i punteggi parziali (con la categoria di conoscenza a cui si riferiscono) da sommare per ottenere il valore finale.

- PEDONE:
  - m: il valore materiale di un pedone è di 100 punti;
  - b: Se il pedone è attaccato e non e' difeso, o e' attaccato da almeno un altro pedone, allora viene assegnata la penalità costante HUNGP; in caso contrario, l'essere semplicemente attaccato da più pezzi di quanti lo difendono comporta una penalità costante ATAKD.
  - k: se il re amico non è più nella casella iniziale, ed il pedone in oggetto si trova nella colonna a, b, c, f, g o h, ad una distanza di al massimo due caselle da esso, allora viene aggiunto un punteggio

PAWNSHIELD, dipendente dallo stadio in cui si trova la partita; se invece il re è ancora nella casella iniziale, ed il pedone in oggetto si trova nella colonna a, b, g o h, nella seconda o terza traversa, allora viene aggiunto sempre il punteggio precedente, ma dimezzato;

- c: se il pedone è centrale (colonna d o e), e si trova ancora nella casella iniziale, allora viene assegnata la penalità costante PEDRNK2B;
- p: in ogni caso viene aggiunto un punteggio dipendente dalla casella in cui si trova il pedone, definito nall'array PawnAdvance[sq] (se il pedone è isolato questo punteggio viene ridotto di un fattore 7/10). Se il pedone è isolato viene assegnata una penalità dipendente dalla colonna in cui si trova, definita nell'array ISOLANI[column]; se invece è doppiato viene assegnata la penalità costante PDOUBLED. Se il pedone non è protetto da altri pedoni, e così anche la prossima casa in cui può andare, allora viene assegnata una penalità dipendente dal numero di pezzi avversari che l'attaccano, definita nell'array BACKWARD[atkpiece]; se inoltre la colonna in cui esso si trova è aperta per l'avversario, o la prossima casella in cui può andare è occupata da un altro pezzo, allora vengono assegnate rispettivamente le penalità costanti PWEAK e PBLOK.
- a: se la posizione è di mediogioco ed i due re sono su lati differenti della scacchiera, allora viene incoraggiato l'attacco di pedoni al re avversario (pawnstorm) assegnando una penalità ai pedoni presenti nello stesso lato dove si trova il re avversario e non utilizzati per l'attacco. La penalità detta dipende dalla traversa in cui si trova il pedone, ed è tanto maggiore quanto più il pedone e' vicino alla sua casella di partenza.
  Se il pedone si trova su una colonna libera da pedoni avversari, vengono assegnati punteggi dipendenti dalle possibilità che ha il pedone di arrivare alla promozione, e dalla traversa in cui esso si trova: se l'avversario ha un pedone che controlla una casella che esso deve attraversare per arrivare a promozione, allora viene aggiunto un valore definito nell'array PassedPawn3[rank] (ridotto di un fattore dipendente dallo stadio in cui si trova la partita); se invece l'avversario controlla o blocca una casella che esso deve attraversare per arrivare a promozione con un pezzo che non è un pedone, oppure ha il re che e' in grado di raggiungerlo prima che esso arrivi a promozione, allora viene aggiunto un valore definito nell'array PassedPawn2[rank] (ridotto dello stesso fattore); se invece non sono vere le due condizioni precedenti, ma l'avversario possiede almeno un pezzo che non è un pedone, allora viene aggiunto un valore definito nell'array PassedPawn1[rank] (sempre ridotto dello stesso fattore); se infine non è vera nessuna delle condizioni precedenti, viene semplicemente aggiunto un valore definito nell'array PassedPawn0[rank].
- r: la presenza di molti pedoni è considerata un fatto positivo, quindi

viene aggiunto un punteggio PawnBonus, dipendente dallo stadio in cui si trova la partita.

- CAVALLO:
  - m: il valore materiale di un cavallo è di 350 punti;
  - b: viene assegnato il punteggio KNIGHTSTRONG, dipendente dallo stadio in cui si trova la partita, quando i pedoni nemici non possono allontanare il cavallo dalla posizione che occupa; inoltre, se il cavallo è attaccato e non difeso, o e' attaccato da un pezzo di valore minore, allora viene assegnata la penalità costante HUNGP; in caso contrario, l'essere semplicemente attaccato da più pezzi di quanti lo difendono comporta una penalità costante ATAKD.
  - x o c: viene assegnato un punteggio dipendente dalla casella in cui si trova il cavallo in oggetto, definito nell'array pknight[sq] (si fa notare che per il cavallo la vicinanza al centro della scacchiera equivale anche ad una maggiore mobilità);
  - k: viene assegnato un punteggio dipendente dalla distanza dal proprio re : più il cavallo è lontano dal re amico e minore è il punteggio assegnato, così da favorire la protezione del proprio re.
  - a: analogamente all'euristica che controlla la sicurezza del proprio re, viene assegnato un punteggio dipendente dalla distanza dal re avversario : più il cavallo è lontano dal re avversario e minore è il punteggio assegnato, così da favorire l'attacco contro di esso . È inoltre assegnato un punteggio che premia il cavallo quando è vicino ai pezzi nemici: tale punteggio è definito da $navv * KNIGHTPOST$, dove $navv$ è il numero di pezzi avversari che si trovano a distanza ≤ 2 dal cavallo in oggetto, mentre KNIGHTPOST è un valore dipendente dallo stadio in cui si trova la partita;
  - r: viene dato un bonus di 5 punti se entrambi i cavalli sono ancora sulla scacchiera;
- ALFIERE:
  - m: il valore materiale di un alfiere è di 355 punti;
  - b: viene assegnato il punteggio BISHOPSTRONG, dipendente dallo stadio in cui si trova la partita, quando i pedoni nemici non possono allontanare l'alfiere dalla posizione che occupa; inoltre, se l'alfiere è attaccato e non difeso, o e' attaccato da un pezzo di valore minore, allora viene assegnata la penalità costante HUNGP; in caso contrario, l'essere semplicemente attaccato da più pezzi di quanti lo difendono comporta una penalità costante ATAKD.
  - x: viene assegnato un punteggio dipendente dalla casella in cui si trova l'alfiere in oggetto, definito nell'array pbishop[sq]; viene inoltre

assegnato un bonus dipendente dal numero di caselle raggiungibili dall'alfiere, definito nell'array BMBLTY[numsq];

- a: viene assegnato un punteggio che premia l'alfiere quando minaccia caselle in prossimità del re avversario (tale punteggio è definito da $sqatk * KATAK$, dove $sqatk$ è il numero di caselle vicine al re avversario che l'alfiere in oggetto può raggiungere, e KATAK è il bonus costante da assegnare per ognuna di esse); un altro che premia l'alfiere quando minaccia i pezzi avversari (tale punteggio è definito da $npin * PINVAL$, dove $npin$ è il numero di pezzi avversari che l'alfiere in oggetto può incontrare lungo il suo percorso, e PINVAL è il bonus costante da assegnare per ognuno di essi); un altro che premia l'alfiere quando può eseguire attacchi di scoperta (tale punteggio è definito da $npc * XRAY$, dove npc è il numero di pezzi amici che l'alfiere in oggetto trova lungo il suo percorso, ed XRAY è il bonus costante da assegnare ad ognuno di essi);
- r: la presenza di un alfiere è considerata un fatto positivo, quindi viene aggiunto un punteggio BishopBonus, dipendente dallo stadio in cui si trova la partita; un ulteriore bonus di 8 punti viene dato se entrambi gli alfieri sono ancora sulla scacchiera;

• TORRE:

- m: il valore materiale di una torre è di 550 punti;
- b: se la torre è attaccata e non difesa, o è attaccata da un pezzo di valore minore, allora viene assegnata la penalità costante HUNGP; in caso contrario, l'essere semplicemente attaccata da più pezzi di quanti la difendono comporta una penalità costante ATAKD.
- x: viene assegnato un bonus dipendente dal numero di caselle raggiungibili dalla torre in oggetto, definito nell'array RMBLTY[numsq]; se la torre si trova su una colonna semiaperta, viene aggiunto un punteggio costante RHOPN; se invece la colonna è completamente aperta, allora il punteggio costante aggiunto è RHOPN + RHOPNX;
- a: viene assegnato un punteggio che premia la torre quando minaccia caselle in prossimità del re avversario (tale punteggio è definito da $sqatk * KATAK$, dove $sqatk$ è il numero di caselle vicine al re avversario che la torre in oggetto può raggiungere, e KATAK è il bonus costante da assegnare per ognuna di esse); un altro che premia la torre quando minaccia i pezzi avversari (tale punteggio è definito da $npin * PINVAL$, dove $npin$ è il numero di pezzi avversari che la torre in oggetto può incontrare lungo il suo percorso, e PINVAL è il bonus costante da assegnare per ognuno di essi); un altro che premia la torre quando può eseguire attacchi di scoperta (tale punteggio è definito da $npc * XRAY$, dove npc è il numero di pezzi amici che la torre in oggetto trova lungo il suo percorso, ed

XRAY è il bonus costante da assegnare ad ognuno di essi);
un bonus di 10 punti viene assegnato se la torre si trova nella settima traversa; se infine la partita è in uno stadio immediatamente successivo all'apertura, viene aggiunto un punteggio tanto minore quanto più è difficile per la torre in oggetto raggiungere il re nemico a partire dalla casella in cui si trova;

- r: la presenza di una torre è considerata un fatto positivo, quindi viene aggiunto un punteggio RookBonus, dipendente dallo stadio in cui si trova la partita;

• REGINA:

- m: il valore materiale della regina è di 1100 punti;
- b: se la regina è attaccata e non difesa, o è attaccata da un pezzo di valore minore, allora viene assegnata la penalità costante HUNGP; in caso contrario, l'essere semplicemente attaccata da più pezzi di quanti la difendono comporta una penalità costante ATAKD.
- a: viene assegnato un bonus di 12 punti se la distanza della regina dal re nemico è minore di 3 caselle; inoltre, se la partita è in uno stadio immediatamente successivo all'apertura, viene aggiunto un punteggio tanto minore quanto più è difficile per la regina raggiungere il re nemico a partire dalla casella in cui si trova;

• RE:

- m: il valore materiale del re è stato fissato in 1200 punti, anche se in effetti non ha effetto sulla valutazione, essendo sicuramente presente per entrambi i giocatori;
- k o c: al re è data una penalizzazione per la vicinanza al centro nel mediogioco, ed un bonus per la vicinanza al centro nel finale. Il valore da aggiungere si ricava combinando linearmente, a seconda dello stadio in cui si trova la partita, il contenuto degli array KingOpening[sq] e KingEnding[sq](riducendo di un fattore 1/2 se il valore del materiale presente nella scacchiera è maggiore di POSLIMIT). L'euristica a cui fa capo questo punteggio è k se il materiale sulla scacchiera ha le potenzialità di minacciare la sicurezza del re, e c altrimenti.
- k: se ci si trova nel mediogioco, allora tramite la funzione KingScan(sq) vengono assegnate penalità nel caso che il re possa essere minacciato da scacchi, o se le caselle vicine al re sono controllate da pezzi avversari (specialmente dalla regina), o se non ci sono pedoni vicino al re; se il re è arroccato, viene aggiunto il punteggio KCASTLD, dipendente dallo stadio in cui si trova la partita; se invece il re non è arroccato ed é già stato mosso viene assegnata la penalità KMOVD,

anch'essa dipendente dallo stadio in cui si trova la partita; vengono infine assegnate le seguenti penalità, sempre dipendenti dallo stato in cui si trova la partita: KHOPN se il re si trova in una colonna senza pedoni amici, o se la colonna adiacente più vicina al bordo della scacchiera è senza pedoni amici; KHOPNX se il re si trova in una colonna senza pedoni avversari, o se la colonna adiacente più vicina al bordo della scacchiera è senza pedoni avversari;

- a: per incentivare il re a sostenere i propri pedoni tenendo sotto controllo quelli avversari, viene assegnata una penalità, dipendente dallo stadio in cui si trova la partita, proporzionale alla distanza esistente tra il re e tutti i pedoni presenti sulla scacchiera. Tale penalità è dimezzata se il valore del materiale presente sulla scacchiera è maggiore di POSLIMIT;
- indipendentemente da qualsiasi euristica, è previsto anche che venga assegnata la penalità costante HUNGP se il re è attaccato e non difeso, o è attaccato da un pezzo di valore minore; in caso contrario, è previsto che l'essere semplicemente attaccato da più pezzi di quanti lo difendono comporti una penalità costante ATAKD.

Infine, la funzione $extra(x)$ è interamente compresa nella euristica b, e prevede l'applicazione della penalità costante HUNGX nel caso che contemporaneamente più di un pezzo del colore x si trovi attaccato e non difeso, o attaccato da un pezzo di valore minore.

3.7 Confronto con Gnuchess 3.1

Il programma Gnuchess è in continua evoluzione, per cui nel corso dei mesi passati ha subito diverse modifiche: in particolare, qui interessano quelle modifiche che differenziano la versione 4.0 patch level 62, utilizzata per il presente lavoro, dalla versione 3.1, usata per gli esperimenti riportati in [29]. Intendiamo innanzitutto sottolineare il fatto che è stata modificata la struttura complessiva di Gnuchess 3.1: il programma, infatti, è stato suddiviso in più moduli per consentire una più facile manutenzione ed agevolare così lo sviluppo, relegando ad un complesso Makefile il compito di ricostruire le funzionalità complessive dell'eseguibile.
Per quanto riguarda le variazioni del codice, i seguenti sono i principali aggiornamenti che hanno portato alla realizzazione della versione 4.0, patch level 62:

- è stata aggiunta la gestione delle trasposizioni di mosse durante la fase di apertura della partita: Gnuchess 3.1 non era in grado di riconoscere eventuali differenze nell'ordine delle mosse eseguite rispetto alle linee di gioco contenute nel libro delle aperture;

- nella versione attuale è possibile ricostruire interamente una partita: ciò non era sempre possibile prima, perché non veniva tenuta traccia del pezzo con cui era eventualmente promosso un pedone;
- vi sono stati ripetuti aggiornamenti nella gestione del tempo: ora il calcolo del tempo è più dinamico, e consente ad esempio un prolungamento della ricerca nel caso di posizioni instabili;
- è stato aggiunto un file (gnuchess.lang), il quale permette la visualizzazione dei messaggi di Gnuchess 4.0 in varie lingue;
- i valori delle variabili che definiscono il peso delle diverse porzioni di conoscenza sono stati più volte aggiustati nel tentativo di ottenere prestazioni migliori da parte del programma;
- è stata introdotta la nuova opzione -T <size> al comando di invocazione di Gnuchess 4.0: essa dà la possibilità di specificare la dimensione desiderata per la tabella delle trasposizioni;
- sono stati aggiunti alcuni comandi all'interfaccia di comunicazione con l'utente, tra i quali material, p, debug e le visualizzazioni delle tabelle di valutazione statica dei pezzi;
- è stata inclusa l'interfaccia per X-windows (Xboard) nella distribuzione di Gnuchess 4.0, con la creazione di una variante ad hoc di Gnuchess 4.0 da utilizzare con essa (gnuchessx);
- per quanto riguarda i miglioramenti della funzione di valutazione statica, è stata introdotta una routine apposita (ScoreK1PK) per l'esame di quelle posizioni in cui siano presenti solamente il re da una parte e re con un pedone dall'altra;
- è stata variata la finestra $\alpha\beta$ prevista in caso di fallimento iniziale della ricerca nell'algoritmo di Aspiration Search: la nuova finestra è quella canonica $]+\infty, -\infty[$.

3.8 Xboard

Gnuchess 4.0 può usufruire dei servizi offerti da una interfaccia grafica (distribuita anch'essa dalla Free Software Foundation), da utilizzare nell'ambiente X-windows: Xboard. Xboard definisce una finestra grafica rappresentante una scacchiera, la quale può servire come interfaccia utente sia per Gnuchess 4.0 che per l'Internet Chess Server (ICS), oppure può essere usata semplicemente per giocare delle partite (o eseguendo le mosse manualmente, o prelevandole da un archivio di partite).
Come interfaccia per Gnuchess 4.0, Xboard consente di giocare una partita contro Gnuchess 4.0, partendo dalle posizioni desiderate ed eventualmente

forzando la macchina a giocare le varianti desiderate, oppure consente di guardare una partita mentre è svolta da due giocatori artificiali.
Come interfaccia per l'ICS, Xboard permette di giocare una partita contro altri utenti dell'ICS, di osservare le partite che essi stanno giocando o di rivedere partite che sono terminate da poco.
Dopo aver lanciato il comando di invocazione, appare una scacchiera i cui pezzi possono essere mossi per mezzo del mouse, premendo l'apposito bottone quando il cursore si trova sulla casella del pezzo che si desidera muovere, trascinando il cursore col bottone premuto fino alla casella dove si vuole posizionare il pezzo, ed infine rilasciando il bottone.
Le opzioni che possono essere inserite al momento dell'invocazione di Xboard differiscono in alcuni casi a seconda dell'uso che si vuole fare dell'interfaccia. Come interfaccia per Gnuchess 4.0, Xboard permette di definire il controllo del tempo e la massima profondità a cui può arrivare Gnuchess 4.0 durante la visita dell'albero di gioco; inoltre dà la possibilità di far giocare due programmi di scacchi posti anche su differenti workstation.
Come interfaccia per l'ICS, invece, Xboard permette di specificare il nome dell'Internet host, o l'indirizzo del chess server, con cui ci si desidera collegare, e di inviare automaticamente alcuni dei messaggi previsti dal protocollo di comunicazione dell'ICS, come "who" per vedere chi è connesso, "games" per vedere quali sono le partite in corso, "match" per sfidare un altro giocatore, ecc.
In ogni caso, sono sempre disponibili le opzioni di caricamento e salvataggio di partite, ed anche le opzioni che regolano la visualizzazione della scacchiera, come -boardSize (definisce la dimensione della scacchiera), -showCoord (mostra le coordinate delle caselle della scacchiera), -flipView (inverte il punto di vista della scacchiera), ecc.
Una volta aperta la finestra grafica, è disponibile un Menu di comandi per l'utilizzazione delle funzionalità previste nell'ambiente in cui ci si trova: ad esempio, durante una partita contro Gnuchess 4.0 possono essere selezionati alcuni dei comandi forniti dallo stesso Gnuchess 4.0 (come Machine white, Force moves, Two machines, ecc.), mentre in una sessione ICS sono disponibili alcuni comandi di ICS (come Decline Draw, Resign, ecc.).
La versione di Xboard distribuita insieme a Gnuchess 4.0 non prevede il suo utilizzo come interfaccia tra Gnuchess 4.0 e l'ICS: le due opzioni in effetti sono mutuamente esclusive. Esiste invece una versione di Xboard, gentilmente concessa da Tim Mann, che consente anche questa possibilità. Useremo la versione ampliata di Xboard per testare le capacità del giocatore a conoscenza distribuita nei confronti degli utenti dell'ICS.

Capitolo 4

Come distribuire la conoscenza?

Per creare un giocatore parallelo come richiesto dal metodo di distribuzione della conoscenza, dobbiamo

1. definire una combinazione di istanze in modo tale che l'unione delle loro euristiche rappresenti l'intera conoscenza di Gnuchess 4.0;

2. stabilire un criterio di selezione per la scelta tra le mosse proposte dalle istanze di quella definitiva.

Il primo problema che ci si pone è quello di determinare l'importanza relativa delle varie porzioni di conoscenza di Gnuchess 4.0, in modo da avere un elemento che possa guidare nelle distribuzione della conoscenza tra le istanze del giocatore parallelo.
Una volta ottenuto l'ordinamento delle euristiche, si cerca di determinare in quale modo è meglio associarle per formare la conoscenza delle istanze, e come combinare le istanze tra loro.
Infine, si esegue una valutazione della bontà di alcuni criteri di selezione, rispetto alle combinazioni di istanze definite precedentemente.

4.1 Ordinamento delle euristiche

4.1.1 Descrizione del metodo

Al fine di determinare l'importanza relativa delle varie porzioni di conoscenza di Gnuchess 4.0, si può sfruttare il lavoro eseguito da Schaeffer [25] relativamente allo studio della conoscenza del giocatore artificiale di scacchi Phoenix. Lo scopo di Schaeffer era quello di determinare quale sottinsieme di conoscenza fosse più importante da dare ad un programma di scacchi, lasciando a disposizione per eventuali aggiunte successive la conoscenza meno importante.

Per questo, egli aveva suddiviso la conoscenza di Phoenix in 8 porzioni (euristiche), alcune coincidenti con quelle di Gnuchess 4.0 (come la sicurezza del re) ed altre specifiche del programma che aveva usato per la sperimentazione (come il punteggio incrementale, consistente nell'assegnazione di un punteggio associato alle mosse eseguite per arrivare alla posizione da valutare).
Schaeffer si proponeva di determinare l'importanza relativa delle porzioni di conoscenza di Phoenix, e lo stesso metodo, in parte modificato, ci può servire per determinare l'importanza relativa delle porzioni di conoscenza di Gnuchess 4.0.
L'esperimento consisteva nel far giocare tra loro versioni di Phoenix contenenti quantità differenti di euristiche, cominciando con le sfide tra una versione che si interessa solamente al materiale, e versioni comprendenti oltre alla valutazione del materiale anche una sola altra euristica. Dopodiché veniva ripetuto il processo acquisendo l'euristica che aveva dato i migliori risultati ed aggiungendo una alla volta le altre.
Ogni sfida consisteva nella disputa di 20 partite, metà col bianco e metà col nero. Il numero di partite era stato scelto in modo da avere un banco di prova sufficiente per ottenere risultati attendibili: questo è confermato da analisi eseguite per il sistema di classificazione ELO [17, 14]. D'altra parte, nei tornei di qualificazione per il campionato del mondo di scacchi si disputano meno di 20 partite, e lo stesso campionato del mondo è di 24 partite.
I risultati avevano determinato un ordinamento delle porzioni di conoscenza, ed avevano anche evidenziato che l'aggiunta di conoscenza non può essere fatta in maniera arbitraria, perché in alcuni casi si ottiene un peggioramento delle prestazioni. In altri termini, ci sono porzioni di conoscenza che sono utili solo quando altre porzioni sono già presenti, risultando negative altrimenti.
A causa del tempo richiesto per eseguire questi esperimenti (per la disputa delle 20 partite di ogni match era impiegata una media di 50 ore), si è pensato di trasportare lo stesso metodo in un altro ambito, che andiamo a presentare.
Da [21] è stato tratto un insieme di 500 posizioni scacchistiche, relative alla fase del mediogioco ed appositamente studiate per testare la qualità di un giocatore: per ciascuna di esse sono note alcune possibili continuazioni, tra le quali si possono trovare le mosse considerate migliori ed anche le mosse da evitare (queste ultime evidenziate col simbolo ?).
Siccome la codifica delle posizioni non contiene tutti i dati, si sono fatte le seguenti ipotesi:

1. ogni volta che re e torri si trovano nella loro casella iniziale, si è supposto che non abbiano ancora mosso, e quindi che a meno di altri impedimenti possa ancora avvenire l'arrocco;

2. non sono ammesse catture en passant.

Chiaramente, queste assunzioni possono essere false per una piccola percentuale di posizioni, ma la loro influenza sul risultato finale è assolutamente irrilevante.
Le posizioni sono codificate secondo la notazione Forsyth: la scacchiera è descritta una riga alla volta, con lettere maiuscole per i pezzi bianchi, lettere minuscole per i pezzi neri e numeri per le caselle vuote. In fondo alla codifica della posizione è indicato il colore a cui tocca muovere, seguito dalle mosse previste, in notazione algebrica, e da eventuali commenti tra parentesi.
Ad esempio, alla seguente codifica corrisponde la posizione in figura ??.

```
  1rq3k1/4rpb1/2p3p1/4p1Pp/1p2P3/pPP2Q1P/P1B2P2/2KR3R/w
                    Pc3c4?,Bc2d3
(Bd3 planning Bc4 is better than c4??, which is "suicide".)
                    Gelfer.99.495
```

Supponendo che le posizioni rappresentino un test sufficientemente significativo, si è pensato di valutare una istanza composta da una combinazione di euristiche osservando quali mosse sceglie nelle varie posizioni, e contando il numero di volte che la scelta ricade su una mossa considerata buona.
Questo metodo di valutazione non è perfetto, in quanto ignora la qualità delle mosse che non coincidono con quelle considerate buone; tuttavia possiede una sua motivazione intuitiva, ed è supportato dai risultati di vari test, come quello eseguito con le posizioni di Bratko-Kopec [9], comprese tra le posizioni esaminate, secondo il quale risulta affermata la forte correlazione dei punteggi ottenuti da giocatori umani e computer con la loro graduatoria nelle classifiche dei tornei. Invece che fare giocare tra loro due istanze in un match di 20 partite, si guarda quale delle due sceglie il maggior numero di mosse buone. In questo modo si riescono a confrontare tra loro le istanze a due a due al solo tempo di valutazione di 500 posizioni.

4.1.2 Risultati dell'esperimento

Una volta definito il metodo per confrontare le istanze tra loro, è stato eseguito l'esperimento, assegnando per la valutazione di ciascuna posizione un tempo di 60 secondi (la valutazione di un' istanza ha richiesto così un tempo totale di circa 8 ore e mezza). Il risultato della valutazione è riassunto nella tabella che segue.

	+p	+a	+c	+b	+r	+k	+x
m	85	83	72	78	64	73	68

m + p	95	90	91	92	85	88
m + p + a	101	96	92	94	92	
m + p + a + c	104	98	94	91		
m + p + a + c + b	106	105	96			
m + p + a + c + b + r	107	100				
m + p + a + c + b + r + k	100					

Ogni riga della tabella fornisce i risultati delle valutazioni eseguite da versioni di Gnuchess 4.0 contenenti in comune le euristiche specificate all'inizio di ogni riga, ed in aggiunta l'euristica indicata in cima alla colonna. Ad esempio, nella terza riga e quarta colonna c'è il risultato della valutazione delle posizioni da parte della versione di Gnuchess 4.0 contenente le euristiche di base "mpa" ed a cui è stata aggiunta l'euristica "b".
La conoscenza totale della versione che ha ottenuto il migliore risultato in una riga diventa la conoscenza comune delle versioni che sono valutate nella riga immediatamente sottostante.
L'ordine di aggiunta delle euristiche, a partire dall'euristica sicuramente più importante, quella della valutazione del materiale (m), è risultato essere m p a c b r k x.
La differenza con l'ordinamento determinato da Schaeffer [25], e mantenuto da Tozzi nel suo lavoro [29], deriva sicuramente dal fatto che, come detto precedentemente, le euristiche valutate sono rappresentate all'interno del programma da porzioni di codice identificate secondo criteri intuitivi dagli autori delle sperimentazioni, e difficilmente confrontabili quando i programmi in esame sono diversi tra loro (come è il caso di Phoenix e Gnuchess 4.0).
I risultati dell'esperimento mostrano in alcuni casi come le interazioni esistenti tra alcune porzioni di conoscenza portino al peggioramento delle prestazioni nonostante l'aumento di conoscenza: per esempio, il numero di posizioni in cui l'istanza "mpac" ha indovinato la mossa migliore è maggiore rispetto a quello di "mpac"+"x". Questo può essere spiegato dal fatto che solo la presenza di alcune euristiche può fornire l'ambiente necessario ad altre per garantire una valutazione efficace delle posizioni: d'altra parte allo stesso risultato è giunto Schaeffer analizzando gli esiti dei suoi tornei. Risulta invece degno di nota il punteggio basso ottenuto dalla funzione di valutazione completa di Gnuchess 4.0: non risulta sempre vera pertanto l'affermazione di Tozzi secondo cui le istanze sono singolarmente più deboli di Gnuchess 4.0.

4.2 Combinazioni di istanze

4.2.1 Criteri di distribuzione

A causa dell'elevato numero di combinazioni di istanze che possono essere definite (il numero di istanze è variabile, e ad ogni istanza può essere associata una tra 2^8 possibili funzioni di valutazione, tante quante sono le diverse combinazioni delle 8 euristiche considerate), ci si vede costretti a circoscrivere la sperimentazione ad un sottinsieme di combinazioni definito secondo metodi empirici.
Innanzitutto, si sceglie per motivi pratici (ed in accordo con quanto detto a pag. 25) di contenere tra 4 e 7 il numero di istanze da combinare per formare il giocatore parallelo.
Restano ora da definire alcuni criteri intuitivi per determinare istanze dall'accorpamento di porzioni di conoscenza, e per unire le istanze a formare un limitato numero di combinazioni. Utilizziamo a tale scopo le indicazioni fornite dal lavoro di Tozzi [29], nel quale vengono mostrati 4 criteri di distribuzione della conoscenza (minima conoscenza, distribuzione bilanciata, distribuzione sbilanciata e distribuzione incrementale). Di questi 4 criteri, però, non consideriamo il primo (minima conoscenza), che prevede di assegnare ad ogni istanza l'euristica più importante insieme ad una sola altra euristica scelta a partire dalle più importanti: questo metodo di distribuzione infatti non rispetta il nostro obiettivo di assicurare che l'unione delle euristiche assegnate a tutte le istanze dia sempre la totalità delle euristiche di Gnuchess 4.0. In sostituzione del criterio di minima conoscenza abbiamo definito un criterio denominato di distribuzione con nucleo comune, che tra gli altri genera un giocatore che soddisfa anche il criterio eliminato.
Vediamo una descrizione dei criteri con la definizione delle combinazioni di istanze ricavate da essi, tenendo sempre presente l'ordinamento delle euristiche scaturito dall'esperimento precedente.

1. distribuzione con nucleo comune: si definisce un nucleo comune a tutte le istanze, composto da quelle euristiche che si sono rivelate essere le più importanti da aggiungere, ed a questo si uniscono le altre euristiche, una per ogni istanza:

```
N=4     mbcpa mcpar mkcpa mxcpa

N=5     mcpa mbpa mpar mkpa mxpa

N=6     mpa mcp mbp mpr mkp mxp

N=7     mp ma mc mb mr mk mx
```

2. distribuzione bilanciata: ad ogni istanza è associato lo stesso numero di euristiche, e compatibilmente con questo vincolo si cerca di distribuire ogni euristica sullo stesso numero di istanze (in tal caso non è importante l'ordinamento delle euristiche, anche se l'euristica più importante, quella del materiale, viene sempre associata a tutte le istanze, per ovvi motivi):

```
N=4     mpar mbxp mbkc mcar

N=5     mpar mbxp mbkc mxar mkcp

N=6     mpar mbxp mbkc mxar mkcp mcar

N=7     mpar mbxp mbkc mxar mkcp mcar mbxk
```

3. distribuzione sbilanciata: una delle istanze ha la stessa funzione di valutazione di Gnuchess 4.0, metà delle altre contiene l'euristica più importante insieme ad un'altra delle più importanti, e le altre istanze contengono tutte le euristiche tranne una tra le meno importanti:

```
N=4     mbxkcpar mp ma mbkcpar

N=5     mbxkcpar mp ma mbkcpar mbxcpar

N=6     mbxkcpar mp ma mc mbkcpar mbxcpar

N=7     mbxkcpar mp ma mc mbkcpar mbxcpar mbxkcpa
```

4. distribuzione incrementale: la prima istanza contiene le euristiche più importanti, ed alle altre istanze viene via via aggiunta l'euristica che si è rivelata essere la prima da aggiungere:

```
N=4     mbcpa mbcpar mbkcpar mbxkcpar

N=5     mcpa mbcpa mbcpar mbkcpar mbxkcpar

N=6     mpa mcpa mbcpa mbcpar mbkcpar mbxkcpar

N=7     mp mpa mcpa mbcpa mbcpar mbkcpar mbxkcpar
```

4.2.2 Valutazione delle combinazioni

Le 16 combinazioni di istanze risultanti possono essere valutate utilizzando le 500 posizioni di [21] nel seguente modo.
Data una combinazione di istanze, si sottopone ogni posizione alle istanze che la compongono, e si contano il numero di volte in cui almeno una delle istanze sceglie una mossa considerata buona (unione) ed il numero di volte in cui tutte le istanze contemporaneamente scelgono una mossa considerata buona (intersezione).
Nell'ipotesi che le 500 posizioni rappresenti un test sufficientemente significativo, il valore di unione ed intersezione ci possono dare le seguenti indicazioni:

- unione: permette di vedere quale combinazione di istanze sia più promettente. Infatti, se si riuscisse ad associare ad ogni combinazione di istanze un criterio di selezione ottimale, tale cioè da privilegiare per ogni posizione sempre l'istanza che ha proposto la mossa corretta, allora l'unione rappresenterebbe il valore del giocatore parallelo derivato;

- intersezione: permette di vedere quale combinazione di istanze sia più sicura. Infatti, indipendentemente dal criterio di selezione utilizzato, sicuramente i giocatori paralleli derivati dalle varie combinazioni di istanze indovinerebbero come minimo questo numero di mosse buone.

In definitiva, unione ed intersezione sono parametri che permettono di paragonare tra loro le combinazioni di istanze.
Andiamo ad eseguire la valutazione delle 16 combinazioni di istanze, assegnando per ognuna delle 500 posizioni un tempo di ricerca di 60 secondi.

```
*****************************
DISTRIBUZIONE CON NUCLEO COMUNE
*****************************

mbcpa mcpar mkcpa mxcpa

union=140, intersection=56
----------------------------------------
mcpa mbpa mpar mkpa mxpa

union=151, intersection=53
----------------------------------------
mpa mcp mbp mpr mkp mxp

union=158, intersection=41
----------------------------------------
mp ma mc mb mr mk mx

union=172, intersection=20
----------------------------------------


***********************
DISTRIBUZIONE BILANCIATA
***********************

mpar mbxp mbkc mcar

union=164, intersection=34
----------------------------------------
mpar mbxp mbkc mxar mkcp

union=176, intersection=26
----------------------------------------
mpar mbxp mbkc mxar mkcp mcar

union=183, intersection=25
----------------------------------------
mpar mbxp mbkc mxar mkcp mcar mbxk

union=186, intersection=24
----------------------------------------
```

```
************************
DISTRIBUZIONE SBILANCIATA
************************

mbxkcpar mp ma mbkcpar

union=159, intersection=43
----------------------------------------
mbxkcpar mp ma mbkcpar mbxcpar

union=172, intersection=41
----------------------------------------
mbxkcpar mp ma mc mbkcpar mbxcpar

union=185, intersection=31
----------------------------------------
mbxkcpar mp ma mc mbkcpar mbxcpar mbxkcpa

union=191, intersection=30
----------------------------------------


*************************
DISTRIBUZIONE INCREMENTALE
*************************

mbcpa mbcpar mbkcpar mbxkcpar

union=136, intersection=72
----------------------------------------
mcpa mbcpa mbcpar mbkcpar mbxkcpar

union=142, intersection=65
----------------------------------------
mpa mcpa mbcpa mbcpar mbkcpar mbxkcpar

union=152, intersection=61
----------------------------------------
mp mpa mcpa mbcpa mbcpar mbkcpar mbxkcpar

union=161, intersection=53
----------------------------------------
```

Come è logico, all'aumentare del numero di istanze l'unione aumenta e l'intersezione diminuisce, indipendentemente dal criterio di distribuzione della conoscenza: infatti più istanze ci sono e maggiori sono le probabilità che una delle istanze scelga la mossa migliore, ma d'altra parte è anche minore la probabilità che tutte scelgano una mossa corretta.
Paragonando le combinazioni con lo stesso numero di istanze, si può dire che in generale ad una combinazione relativamente sicura (quindi con una alta intersezione) fa riscontro una potenzialità limitata (quindi un'unione minore).
Dai risultati di questo esperimento si può evincere che i criteri di distribuzione bilanciata e sbilanciata sono più o meno equivalenti, e che nell'ordine sono migliori del criterio di distribuzione con nucleo comune e del criterio di distribuzione incrementale.
L'esito dell'esperimento eseguito trova riscontro nei risultati ottenuti da Tozzi: escludendo il criterio di distribuzione con nucleo comune, viene sostanzialmente confermata la maggiore efficacia dei criteri di distribuzione bilanciata e sbilanciata, e la scarsa validità del criterio di distribuzione incrementale. Vi è comunque una discordanza nella valutazione della combinazione di istanze derivata sia con la distribuzione con minima conoscenza di Tozzi che con la nostra distribuzione con nucleo comune (la combinazione mp ma mc mb mr mk mx): secondo Tozzi questa combinazione di istanze fornisce i migliori risultati, mentre nel nostro esperimento non raggiunge certamente una valutazione particolarmente elevata. Questo può essere parzialmente spiegato dalla diversa versione del programma Gnuchess usata, e dalla differente identificazione delle euristiche all'interno del codice; probabilmente, però, deriva soprattutto dal fatto che le valutazioni eseguite da Tozzi non tengono conto della presenza di mosse considerate deboli (evidenziate con un punto interrogativo) all'interno dell'insieme di mosse fornite per ognuna delle 500 posizioni da valutare.

4.3 Esame dei criteri di selezione

4.3.1 Criteri di selezione

Per definire il giocatore parallelo è necessario affiancare alla combinazione delle istanze un criterio di selezione: la sua importanza è testimoniata dai risultati dell'esperimento precedente,secondo i quali le prestazioni di un giocatore parallelo possono variare notevolmente, passando ad esempio dal valore dell' intersezione (in caso di criterio pessimo) a quello dell'unione (in caso di criterio ottimo).
Esaminiamo, sempre con l'ausilio delle 500 posizioni, le prestazioni dei giocatori paralleli ricavati affiancando alle 16 combinazioni di istanze viste prima i seguenti criteri di selezione, tratti dal lavoro di Tozzi [29]:

1. a maggioranza generalizzata con pesi costanti: viene assegnato ad ogni istanza un peso costante. Ogni volta che il giocatore parallelo deve

selezionare una mossa, associa ad ogni mossa proposta la sommatoria dei pesi delle istanze che l'hanno suggerita, e poi seleziona quella il cui peso risultante è maggiore. Abbiamo considerato un solo criterio per la definizione dei pesi delle istanze:

- criterio "pesi": il peso costante dato alle istanze dipende dalla funzione di valutazione loro associata. Dopo avere definito un peso per ogni euristica, si calcola il peso di una generica istanza sommando tra loro i pesi delle euristiche che ne determinano la funzione di valutazione. Questo si differenzia dal metodo di assegnamento dei pesi alle istanze descritto in [29], secondo il quale il peso di una istanza è definito dalla media dei pesi delle euristiche che la determinano: infatti si ritiene di non dovere sfavorire istanze che valutano più euristiche. Ad esempio, se m è l'euristica di peso maggiore, secondo Tozzi l'aggiunta di un'altra euristica ad essa ne diminuisce il peso, mentre in realtà si deve ritenere che l'aggiunta di euristiche ad una istanza la rendano più efficace nella valutazione dell'albero.
  I pesi associati alle euristiche sono tratti direttamente dal lavoro di Tozzi, considerando però l'ordinamento ricavato nell'esperimento precedente: essi sono $m \Rightarrow 30, p \Rightarrow 21, a \Rightarrow 15, c \Rightarrow 10, b \Rightarrow 8, r \Rightarrow 6, k \Rightarrow 3, x \Rightarrow 1$.

2. a maggioranza generalizzata con pesi variabili: è analogo al criterio precedente, ma prevede che il peso associato ad ogni istanza vari nel corso della partita. Vediamo alcuni criteri per la definizione dei pesi delle istanze:

   - criterio "depth": il peso dato ad un'istanza è funzione della profondità di ricerca raggiunta (d). Per consentire di distinguere tra due istanze che hanno raggiunto la stessa profondità, viene anche considerato il numero nm di mosse esplorate completamente al top-level. Il peso è dunque $p = k * d + nm$, dove k è una costante sufficientemente grande da rendere nm ininfluente nel caso che le profondità raggiunte risultino diverse (si pone $k = 100$).
   - criterio "relativo1": il peso dato ad un'istanza dipende dall'incremento di valore locale che la mossa proposta dall'istanza induce nella posizione attuale. In pratica, viene calcolata la differenza tra il valore minimax della mossa proposta e la valutazione statica della posizione attuale, entrambe determinate rispetto alla funzione di valutazione associata all'istanza.
   - criterio "relativo1 dem": è praticamente un criterio a maggioranza semplice (detto anche democratico), in cui la discriminazione tra mosse con lo stesso peso viene eseguita grazie al criterio "relativo1". Questo è ottenuto aggiungendo al peso ricavato analogamente al

criterio "relativo1" un valore talmente elevato (10000) da rendere marginale l'incremento di valore indotto dalla mossa proposta.

- criterio "relativo2": per determinare il peso di un'istanza viene eseguita un'analisi globale dell'incremento di valore che la mossa proposta dall'istanza induce nella posizione attuale. Diversamente dal criterio "relativo1", in questo caso entrano in gioco anche le altre istanze che compongono il giocatore parallelo: se Pa è la posizione attuale, e Pf è la posizione che conclude la variante principale selezionata dall'istanza, allora il peso è dato da

$$\sum_i (fvs_i(Pf) - fvs_i(Pa))$$

dove fvs_i è la funzione di valutazione statica associata all'i-esima istanza.

3. visita selettiva a posteriori dell'albero di gioco: viene eseguita una nuova valutazione della posizione, considerando però al primo livello solo le mosse proposte dalle istanze che compongono il giocatore parallelo, ed utilizzando la funzione di valutazione completa di Gnuchess 4.0. In questo caso il criterio è chiamato "re_search", e viene realizzato assegnando alla ricerca successiva lo stesso tempo impiegato dalle istanze per la proposta della loro mossa.

4.3.2 Valutazione dei criteri di selezione

Nei risultati dell'esperimento che mostriamo di seguito sono indicati il numero totale di mosse buone (best), il numero totale di mosse sconsigliate (worst) ed il numero di altre mosse (other) che i giocatori paralleli hanno selezionato per tutte le 500 posizioni (best+worst+other=500).

```
*****************************
DISTRIBUZIONE CON NUCLEO COMUNE
*****************************

mbcpa mcpar mkcpa mxcpa

CRITERIO pesi:          best=101, worst=10, other=389
CRITERIO depth:         best=103, worst=10, other=387
CRITERIO relativo1:     best=92,  worst=11, other=397
CRITERIO relativo1 dem: best=101, worst=10, other=389
CRITERIO relativo2:     best=83,  worst=10, other=407
CRITERIO re_search:     best=103, worst=11, other=386
-------------------------------------------

mcpa mbpa mpar mkpa mxpa

CRITERIO pesi:          best=98,  worst=11, other=391
CRITERIO depth:         best=99,  worst=12, other=389
CRITERIO relativo1:     best=99,  worst=12, other=389
CRITERIO relativo1 dem: best=94,  worst=11, other=395
CRITERIO relativo2:     best=92,  worst=10, other=398
CRITERIO re_search:     best=98,  worst=12, other=390
-------------------------------------------

mpa mcp mbp mpr mkp mxp

CRITERIO pesi:          best=93,  worst=12, other=395
CRITERIO depth:         best=98,  worst=12, other=390
CRITERIO relativo1:     best=88,  worst=12, other=400
CRITERIO relativo1 dem: best=91,  worst=12, other=397
CRITERIO relativo2:     best=80,  worst=10, other=410
CRITERIO re_search:     best=98,  worst=13, other=389
-------------------------------------------

mp ma mc mb mr mk mx

CRITERIO pesi:          best=86,  worst=8,  other=406
CRITERIO depth:         best=83,  worst=8,  other=409
CRITERIO relativo1:     best=72,  worst=7,  other=421
CRITERIO relativo1 dem: best=84,  worst=9,  other=407
CRITERIO relativo2:     best=66,  worst=4,  other=430
CRITERIO re_search:     best=97,  worst=12, other=391
-------------------------------------------
```

```
************************
DISTRIBUZIONE BILANCIATA
************************

mpar mbxp mbkc mcar

CRITERIO pesi:          best=89,  worst=12, other=399
CRITERIO depth:         best=89,  worst=13, other=398
CRITERIO relativo1:     best=79,  worst=11, other=410
CRITERIO relativo1 dem: best=87,  worst=13, other=400
CRITERIO relativo2:     best=81,  worst=10, other=409
CRITERIO re_search:     best=96,  worst=12, other=392
---------------------------------------------

mpar mbxp mbkc mxar mkcp

CRITERIO pesi:          best=92,  worst=14, other=394
CRITERIO depth:         best=98,  worst=14, other=388
CRITERIO relativo1:     best=84,  worst=8,  other=408
CRITERIO relativo1 dem: best=91,  worst=12, other=397
CRITERIO relativo2:     best=87,  worst=9,  other=404
CRITERIO re_search:     best=106, worst=13, other=381
---------------------------------------------

mpar mbxp mbkc mxar mkcp mcar

CRITERIO pesi:          best=89,  worst=11, other=400
CRITERIO depth:         best=93,  worst=14, other=393
CRITERIO relativo1:     best=80,  worst=8,  other=412
CRITERIO relativo1 dem: best=89,  worst=10, other=401
CRITERIO relativo2:     best=89,  worst=10, other=401
CRITERIO re_search:     best=104, worst=12, other=384
---------------------------------------------

mpar mbxp mbkc mxar mkcp mcar mbxk

CRITERIO pesi:          best=88,  worst=13, other=399
CRITERIO depth:         best=91,  worst=13, other=396
CRITERIO relativo1:     best=75,  worst=9,  other=416
CRITERIO relativo1 dem: best=88,  worst=13, other=399
CRITERIO relativo2:     best=81,  worst=9,  other=410
CRITERIO re_search:     best=102, worst=12, other=386
---------------------------------------------
```

```
************************
DISTRIBUZIONE SBILANCIATA
************************

mbxkcpar mp ma mbkcpar

CRITERIO pesi:          best=100, worst=9, other=391
CRITERIO depth:         best=99,  worst=9, other=392
CRITERIO relativo1:     best=91,  worst=7, other=402
CRITERIO relativo1 dem: best=101, worst=9, other=390
CRITERIO relativo2:     best=87,  worst=6, other=407
CRITERIO re_search:     best=107, worst=9, other=384
---------------------------------------------

mbxkcpar mp ma mbkcpar mbxcpar

CRITERIO pesi:          best=100, worst=11, other=389
CRITERIO depth:         best=97,  worst=11, other=392
CRITERIO relativo1:     best=91,  worst=7,  other=402
CRITERIO relativo1 dem: best=96,  worst=10, other=394
CRITERIO relativo2:     best=96,  worst=6,  other=398
CRITERIO re_search:     best=107, worst=9,  other=384
---------------------------------------------

mbxkcpar mp ma mc mbkcpar mbxcpar

CRITERIO pesi:          best=97,  worst=11, other=392
CRITERIO depth:         best=100, worst=11, other=389
CRITERIO relativo1:     best=85,  worst=6,  other=409
CRITERIO relativo1 dem: best=93,  worst=11, other=396
CRITERIO relativo2:     best=86,  worst=7,  other=407
CRITERIO re_search:     best=109, worst=9,  other=382
---------------------------------------------

mbxkcpar mp ma mc mbkcpar mbxcpar mbxkcpa

CRITERIO pesi:          best=100, worst=10, other=390
CRITERIO depth:         best=102, worst=11, other=387
CRITERIO relativo1:     best=87,  worst=7,  other=406
CRITERIO relativo1 dem: best=100, worst=11, other=389
CRITERIO relativo2:     best=90,  worst=7,  other=403
CRITERIO re_search:     best=109, worst=10, other=381
---------------------------------------------
```

```
*************************
DISTRIBUZIONE INCREMENTALE
*************************

mbcpa mbcpar mbkcpar mbxkcpar

CRITERIO pesi:          best=111, worst=9,  other=380
CRITERIO depth:         best=112, worst=8,  other=380
CRITERIO relativo1:     best=103, worst=9,  other=388
CRITERIO relativo1 dem: best=111, worst=7,  other=382
CRITERIO relativo2:     best=97,  worst=7,  other=396
CRITERIO re_search:     best=105, worst=10, other=385
-------------------------------------------

mcpa mbcpa mbcpar mbkcpar mbxkcpar

CRITERIO pesi:          best=113, worst=8,  other=379
CRITERIO depth:         best=113, worst=8,  other=379
CRITERIO relativo1:     best=105, worst=9,  other=386
CRITERIO relativo1 dem: best=112, worst=8,  other=380
CRITERIO relativo2:     best=95,  worst=7,  other=398
CRITERIO re_search:     best=108, worst=10, other=382
-------------------------------------------

mpa mcpa mbcpa mbcpar mbkcpar mbxkcpar

CRITERIO pesi:          best=113, worst=10, other=377
CRITERIO depth:         best=114, worst=8,  other=378
CRITERIO relativo1:     best=104, worst=8,  other=388
CRITERIO relativo1 dem: best=111, worst=8,  other=381
CRITERIO relativo2:     best=100, worst=7,  other=393
CRITERIO re_search:     best=105, worst=11, other=384
-------------------------------------------

mp mpa mcpa mbcpa mbcpar mbkcpar mbxkcpar

CRITERIO pesi:          best=112, worst=10, other=378
CRITERIO depth:         best=109, worst=10, other=381
CRITERIO relativo1:     best=104, worst=7,  other=389
CRITERIO relativo1 dem: best=107, worst=10, other=383
CRITERIO relativo2:     best=100, worst=6,  other=394
CRITERIO re_search:     best=106, worst=11, other=383
-------------------------------------------
```

Da questi dati si evince che ci sono 3 criteri nettamente migliori degli altri: sono i criteri "pesi", "depth" e "re_search". Degli altri criteri, l'unico che ha parzialmente retto il confronto è "relativo1 dem", mentre sia "relativo1" che "relativo2" hanno avuto valutazioni nettamente sotto la media.
L'esito di questo esperimento consente un giudizio più ottimista rispetto a quello formulato da Tozzi, i cui risultati sono da egli stesso considerati deludenti in quanto solo un criterio è riuscito a migliorare le prestazioni di Gnuchess 4.0. Nel nostro caso, invece, ai giocatori risultanti dai criteri di selezione è stato assegnato diverse volte un valore superiore a quello assegnato a Gnuchess 4.0 (100), ed in alcune circostanze l'incremento è stato anche consistente (ad esempio, diversi giocatori paralleli hanno indovinato ben 113 mosse buone, ed uno addirittura 114).
In conclusione, abbiamo determinato due criteri di distribuzione (bilanciata e sbilanciata) che consentono di ottenere giocatori paralleli con buone potenzialità, ed inoltre disponiamo di tre criteri di selezione ("pesi", "depth" e "re_search") che sembrano garantire delle promettenti prestazioni. Vedremo più avanti una valutazione dei giocatori che scaturiscono dalla combinazione di questi criteri.
Cercheremo inoltre di esaminarne il comportamento al variare di alcuni dei parametri che abbiamo definito in maniera intuitiva nell'esecuzione degli esperimenti visti. In particolare, tenteremo di stabilire quali pesi da associare alle euristiche permettono di ottenere i migliori risultati attraverso l'esecuzione di un algoritmo genetico. Inoltre, vedremo come si comporta un giocatore parallelo con criterio di selezione "re_search" al variare del rapporto tra il tempo assegnato alla selezione parallela ed il tempo assegnato alla ricerca successiva.

Capitolo 5

Gnupar

Si vuole ricavare usando Network C-Linda un giocatore che realizzi la distribuzione della conoscenza. Sfruttando le potenzialità offerte dal modello Linda di programmazione parallela, definiamo un'architettura flessibile in cui si prevede un numero variabile di istanze, parametriche rispetto alla funzione di valutazione, ognuna delle quali al momento della selezione della mossa esegue una visita autonoma dell'albero di gioco e determina quella che ritiene essere la mossa migliore secondo il proprio punto di vista. Quando tutte le istanze hanno scelto la propria mossa, la comunicano ad un processo coordinatore, il quale provvede ad applicare un criterio di selezione per decidere quale deve essere la mossa definitiva che il giocatore parallelo comunicherà all'avversario. Il giocatore parallelo risultante, realizzato riusando il codice di Gnuchess 4.0, viene chiamato Gnupar.

5.1 Architettura generale

Per realizzare la struttura che abbiamo in mente è necessario un processo coordinatore, che comunichi alle istanze il momento in cui devono eseguire la scelta della propria mossa, e che raccolga i vari risultati per decidere la mossa definitiva: il modello di programmazione parallela che più si adatta a queste esigenze è il Master-Worker.
In un programma di tipo Master-Worker, un processo Master inizializza la computazione e crea un insieme di processi Worker identici, ognuno dei quali è in grado di eseguire i lavori che saranno assegnati dal Master. Il compito dei Worker è quello di attendere un lavoro da eseguire, nel momento in cui arriva eseguirlo, e poi attendere il successivo.
Quando il Master non ha più lavori da fare eseguire, allora cancella i Worker man mano che terminano il loro lavoro, dopodiché il programma finisce. Si fa notare che non è importante il numero dei Worker creati, basta che ce ne sia almeno uno (il numero ottimale di Worker dipende in effetti dal carico di lavoro che deve essere eseguito) [10].
Si fa notare però che, a differenza del modello generale, nel nostro caso i

Worker da creare eseguono sì lo stesso lavoro di visita dell'albero di gioco, ma non risultano identici tra loro, in quanto ognuno possiede una propria particolare conoscenza del gioco: a causa di ciò, ed a causa del fatto che il Master potrebbe avere bisogno di sapere, per l'applicazione del criterio di selezione, secondo quale conoscenza sono state scelte le mosse proposte, si rende necessario contraddistinguere ogni Worker con un identificatore.
Inoltre, siccome nel nostro caso il Master non deve distribuire con continuità i lavori ai Worker, ma richiede ogni volta una risposta ai lavori precedenti prima di assegnare i nuovi, allora vi è per esso un lungo periodo improduttivo, corrispondente all'intervallo di tempo compreso tra la comunicazione ai Worker dell'inizio della loro selezione e la ricezione delle mosse proposte: questo tempo può essere produttivamente impiegato dal Master se esso stesso si attribuisce una porzione di conoscenza ed esegue la propria visita dell'albero di gioco, ricavando una mossa da unire a quelle proposte dai Worker.
Posto che è il Master a gestire tutte le informazioni relative alla partita in corso, quale conoscenza devono avere i Worker per potere eseguire il loro lavoro?
Ci sono due possibilità:

- i Worker non sanno niente della partita in corso: quando sono chiamati ad eseguire un lavoro, il Master comunica loro tutti i dati necessari per eseguire la ricerca (la posizione insieme ai dati ad essa connessi);

- i Worker mantengono una loro copia locale dell'andamento della partita in corso, ed il Master comunica semplicemente l'ordine di iniziare la scelta locale.

A causa del costo di comunicazione che comporterebbe una soluzione del primo tipo, si è preferito realizzare una distribuzione dei dati rilevanti della partita, assegnando ai Worker una rappresentazione locale dello stato del gioco, comprendente la distribuzione dei pezzi, lo stato dei flag di controllo, ecc. La conformità degli stati dei Worker con quello del Master viene garantita, al termine di ogni selezione, dall'invio ai Worker da parte del Master della mossa effettivamente selezionata dal giocatore parallelo, così che essi possano provvedere ad aggiornare conseguentemente i propri dati locali. In occasione di questa comunicazione ha luogo la sincronizzazione delle istanze prima dell'inizio della selezione successiva.
Il controllo del tempo, invece, viene amministrato solamente dal Master, il quale ne gestisce la modalità ed i limiti stabiliti. È dunque il Master che si preoccupa di calcolare il tempo parziale da dedicare alla successiva visita dell'albero di gioco, ed a comunicarlo ai Worker prima dell'inizio della loro ricerca locale: il tempo di ricerca, in questo modo, risulta più o meno lo stesso per tutti i Worker (e per il Master stesso), così da ridurre il periodo improduttivo dovuto all'attesa delle mosse proposte da parte del Master.

5.2 Struttura del Master

In base a queste considerazioni, assegnamo al Master il compito di realizzare il ciclo di base del nostro programma di scacchi, come evidenziato in figura 5.1.

Le uniche differenze rispetto al ciclo di base di Gnuchess 4.0 (vedere figura 3.1) sono l'inizializzazione e la terminazione dei Worker, e la diversa gestione della selezione della mossa e dell'inserimento dei comandi (o della mossa dell'avversario), i quali devono necessariamente tenere conto dell'interscambio di informazioni tra Master e Worker.

5.2.1 Selezione della mossa definitiva

Il cuore di Gnupar è rappresentato dalla procedura di selezione della mossa da parte del giocatore parallelo (Par_SelectMove): è durante la sua esecuzione, infatti, che viene concentrata la maggiore utilizzazione delle risorse a disposizione.
Insieme alla ricerca nell'albero di gioco locale, il Master deve coordinare le operazioni di selezione della mossa da parte dei Worker, raccogliendo al termine l'insieme di mosse proposte, ed estraendo da queste, per mezzo del criterio di selezione previsto, la mossa da eseguire e comunicare prima agli stessi Worker, per il loro aggiornamento, e poi all'avversario. Queste operazioni comportano l'integrazione dell'omologa procedura di Gnuchess 4.0 (SelectMove), descritta in figura 3.2, con opportune ruotine di comunicazione ed eventuale sincronizzazione coi Worker; in aggiunta, viene gestito il ricevimento delle mosse proposte dai Worker e l'applicazione del criterio di selezione (vedere la figura 5.2).

Compito del Master è anche quello di gestire il libro delle aperture, evitando di assegnare un nuovo lavoro ai Worker (e a se stesso) se trovasse in esso la prossima mossa da eseguire.
Per quanto riguarda l'hash file, invece, si ritiene di non dotare il giocatore di tale euristica, in quanto

- verrebbe ritardata l'elaborazione a causa di possibili richieste di accesso ad esso in concorrenza tra le istanze;
- il valore inserito nell'hash file sarebbe fortemente influenzato dalla porzione di conoscenza dell'istanza che l'ha determinato.

Per maggiori particolari sull'interazione tra Master e Worker durante la selezione della mossa, si rimanda al capitolo 5.4.

5.2.2 L'interprete dei comandi

Il ciclo eseguito dalla procedura che realizza l'interfaccia tra il giocatore artificiale e l'utente (Par_InputCommand), è analogo a quello di Gnuchess 4.0

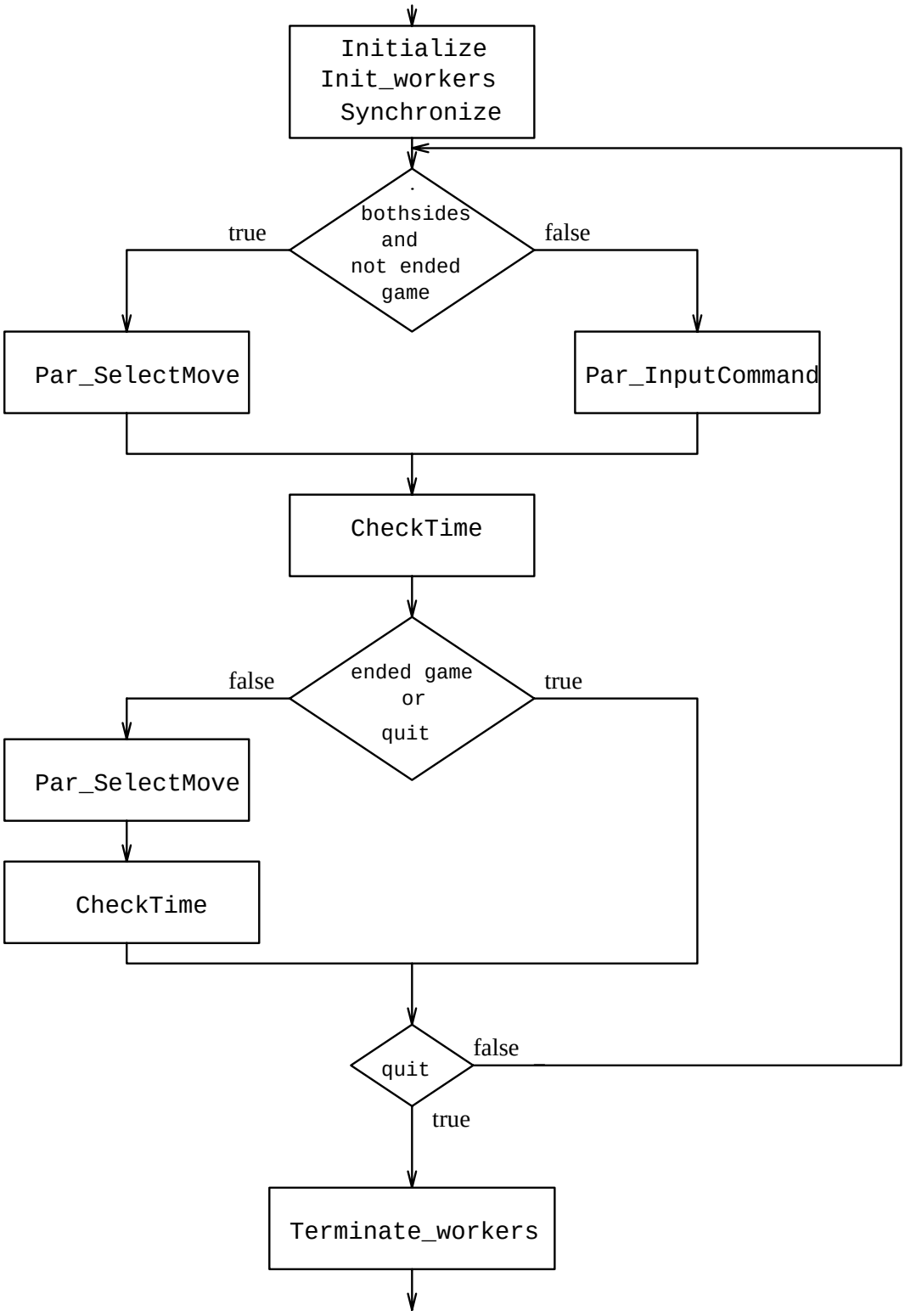

Figura 5.1: Ciclo di base del Master

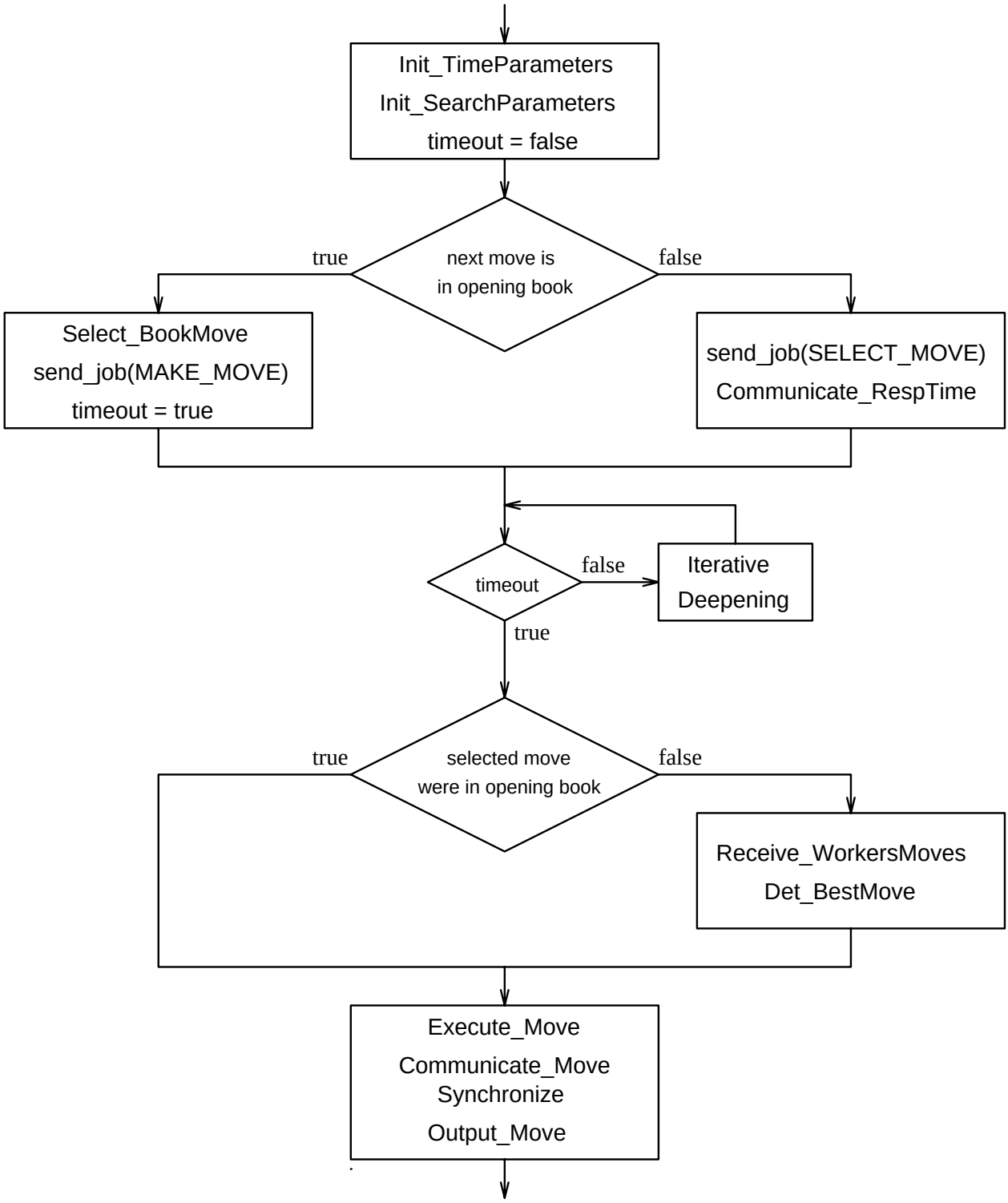

Figura 5.2: Par_SelectMove

(vedere figura 3.3), ma cambia la gestione di alcuni comandi, mentre altri devono essere eliminati perché senza senso in un giocatore parallelo (ad esempio, il comando post, che nel sequenziale visualizza alcuni dati relativi alla ricerca nell'albero man mano che vengono determinati, perde di significato quando vengono eseguite contemporaneamente più ricerche in alberi di gioco distinti).

Nella figura 5.3 viene evidenziato in particolare che nel comando both la procedura di selezione della mossa deve tenere conto dello scambio di informazioni tra Master e Worker, e che al riconoscimento di una mossa valida viene attivato un protocollo che consente di comunicare la mossa acquisita a tutti i Worker, così da permettere loro di aggiornarsi sullo stato effettivo del giocatore parallelo.

5.3 Struttura dei Worker

Il Worker è un semplice esecutore di lavori inviati dal Master, con una propria funzione di valutazione ed una struttura dati locale in cui tenere traccia dello stato della partita.
Al momento della sua creazione, vengono inizializzati i dati locali, e vengono ricevute le comunicazioni del Master riguardanti le opzioni eventualmente immesse dall'utente, come ad esempio dimensione e modalità di accesso per la tabella delle trasposizioni, oppure altri parametri relativi alla ricerca nell'albero di gioco.
Una volta che il Worker è stato inizializzato e si è sincronizzato col Master, ha inizio il ciclo che consiste nella ricezione di un lavoro e nella sua esecuzione. Il ciclo termina quando il lavoro ricevuto è quello di terminazione (QUIT).
I lavori principali che esegue un Worker sono quelli di selezione di una mossa (SELECT_MOVE) e di esecuzione della mossa definitiva (MAKE_MOVE): nella figura 5.4 vengono mostrate le operazioni che svolge il Worker per la loro realizzazione.

5.4 Interazione Master–Worker

Vediamo alcuni esempi su come interagiscono tra loro il Master ed i Worker nella fase di selezione della mossa del giocatore parallelo (Par_SelectMove).

5.4.1 Scelta dal libro delle aperture

Inizialmente (vedere figura 5.5), i Worker sono in attesa del prossimo lavoro da eseguire, mentre il Master inizia l'esecuzione della procedura di selezione della mossa del giocatore parallelo.
Il Master inizializza i parametri temporali e di ricerca, e poi va a controllare

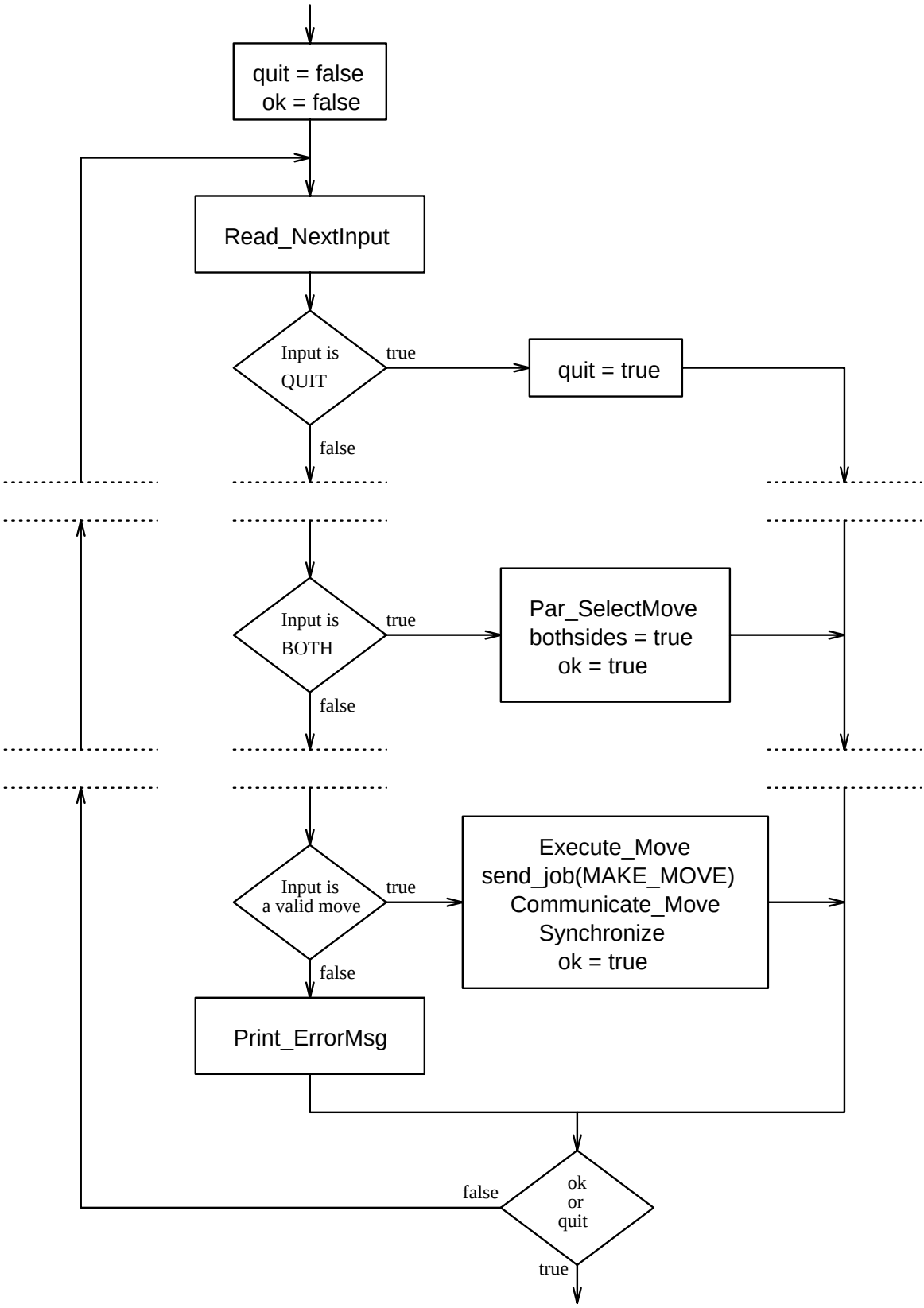

Figura 5.3: Par_InputCommand

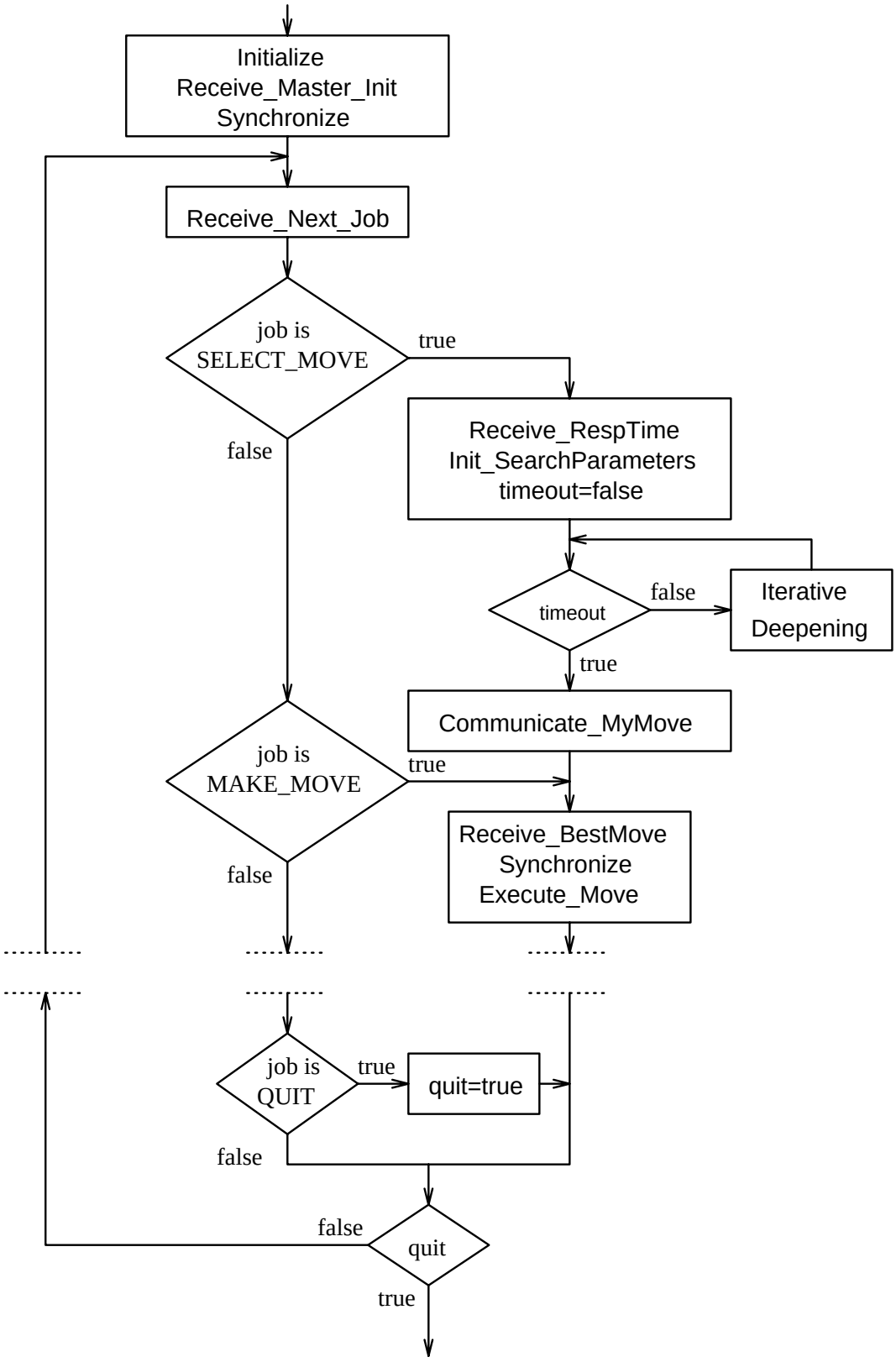

Figura 5.4: Ciclo del Worker

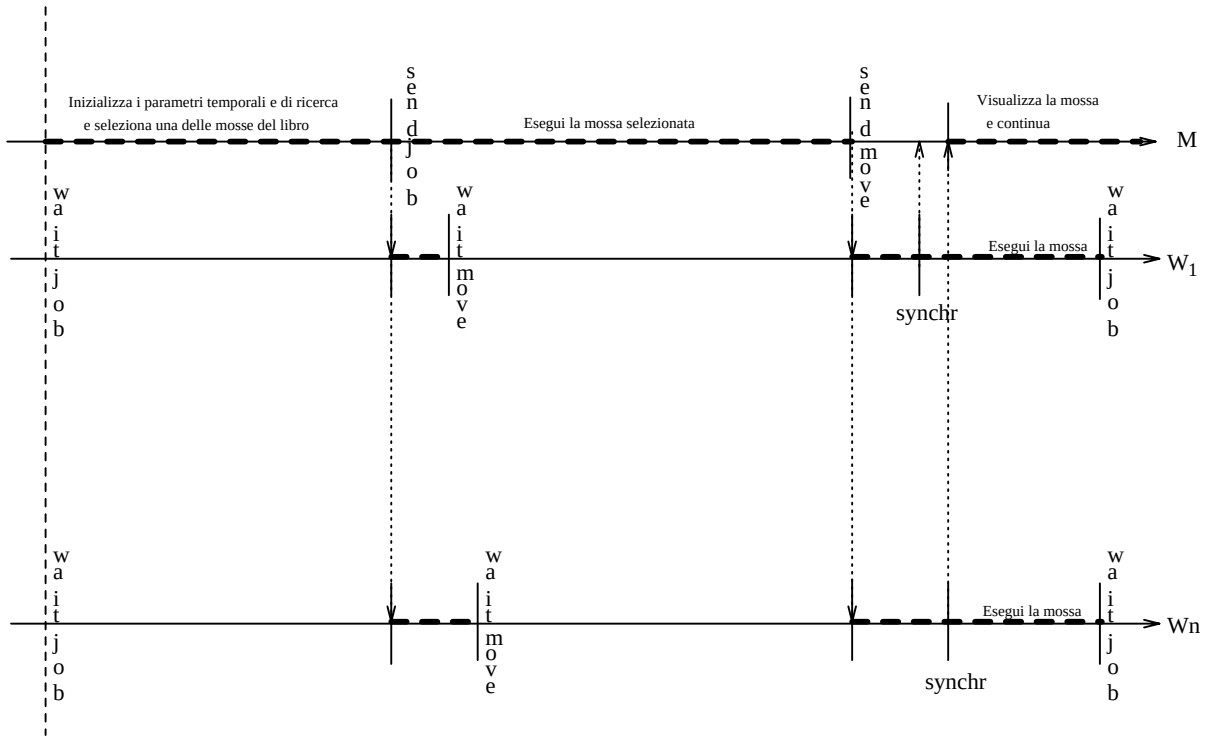

Figura 5.5: Scelta dal libro delle aperture

se le mosse eseguite finora corrispondono ad una delle linee di gioco contenute nel libro delle aperture. Si fa notare che questa sequenza di operazioni deriva dal codice di Gnuchess 4.0, e quindi è rimasta tale in rispetto all'impostazione originaria del programma, anche se in effetti sarebbe più logico l'ordine inverso, essendo inutile l'inizializzazione dei parametri nel caso che la mossa venisse scelta dal libro delle aperture.
Supponiamo che nell'archivio si trovi una linea di gioco coincidente con quella attuale: il Master seleziona in tal caso una delle mosse previste in risposta, e poi comunica ai Worker che il tipo di lavoro che devono compiere è quello di esecuzione della mossa che verrà loro trasmessa (MAKE_MOVE).
Dopo aver comunicato il tipo di lavoro, il Master esegue immediatamente la mossa selezionata, quindi la trasmette ai Worker, mettendosi poi in attesa della conferma che tutti i Worker l'abbiano ricevuta. I Worker, dopo avere ricevuto la comunicazione che il tipo di lavoro da fare è l'esecuzione della mossa scelta dal giocatore parallelo, si mettono in attesa che il Master trasmetta la stessa; al ricevimento della mossa inviano al Master la conferma della ricezione e poi la eseguono, aggiornando in questo modo i propri dati locali conformemente alla situazione globale. A questo punto i Worker hanno finito il loro lavoro, e si mettono in attesa del prossimo.
Quando il Master ha ricevuto tutte le conferme dei Worker, si ha la sincronizzazione: dopo avere riferito all'avversario la mossa selezionata, esso può passare ad eseguire la procedura successiva.

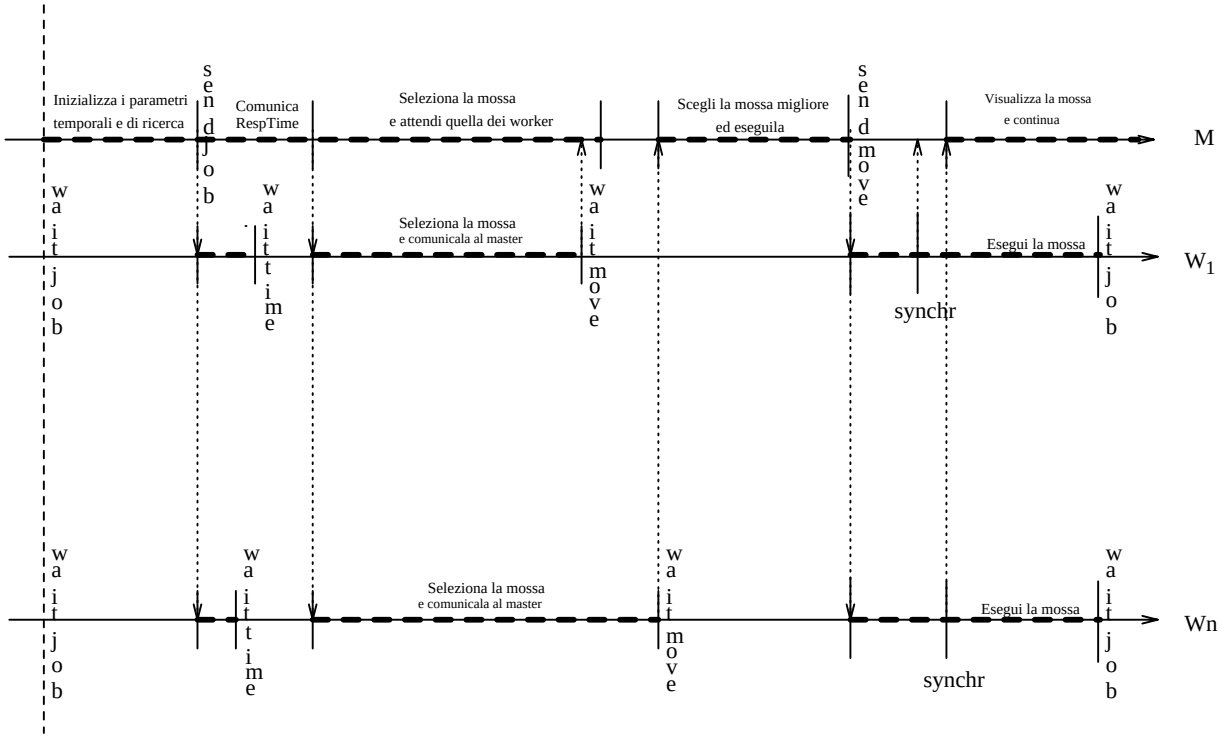

Figura 5.6: Scelta con visita dell'albero di gioco

5.4.2 Scelta normale

Fino al controllo nel libro delle aperture le operazioni eseguite dal Master sono le stesse del caso precedente, ed anche i Worker rimangono ugualmente inattivi in attesa della ricezione di un lavoro.
Supponiamo ora che il Master non trovi nel libro delle aperture una linea di gioco coincidente con quella attuale (vedere figura 5.6): in questo caso esso comunica ai Worker che il tipo di lavoro che devono compiere è quello di selezionare la mossa da proporre (SELECT_MOVE).
Subito dopo avere comunicato il tipo di lavoro, il Master trasmette ai Worker il tempo che essi devono dedicare alla selezione della loro mossa, ed inizia immediatamente la selezione della propria mossa. I Worker, dopo avere ricevuto la comunicazione che il tipo di lavoro da fare è la selezione della mossa da proporre, si mettono in attesa che il Master trasmetta il tempo da dedicare alla selezione: al suo ricevimento iniziano anch'essi la selezione.
Questo è il momento di maggiore utilizzazione delle risorse, in quanto tutte le istanze stanno lavorando contemporaneamente alla ricerca di una mossa.
Quando il Master termina la propria selezione, si pone in attesa delle mosse proposte dai Worker; i Worker, invece, appena terminano la loro selezione trasmettono al Master la mossa che hanno scelto, e si mettono in attesa della mossa definitiva decisa dal giocatore parallelo.
A questo punto la situazione è analoga a quella che conclude l'esempio precedente: il Master esegue la mossa finale, quindi la trasmette ai Worker

ed attende la sincronizzazione; i Worker, al ricevimento della mossa, prima si sincronizzano col Master, e poi la eseguono, mettendosi infine in attesa del prossimo lavoro. Al termine della sincronizzazione, il Master riferisce la mossa finale all'avversario e passa ad eseguire la procedura successiva.

5.5 Funzioni di valutazione parametriche

Dalla decomposizione logica della funzione di valutazione di Gnuchess 4.0 nelle 8 euristiche di cui abbiamo parlato precedentemente, si ricava un metodo di generazione di funzioni [29] basato sulla identificazione delle varie porzioni di conoscenza all'interno del codice, e sulla combinazione di una parte di esse.
Le 8 porzioni di conoscenza sono quelle indicate nel capitolo 3.6.1:

- m = MATERIALE
- b = BUONA SISTEMAZIONE DEI PEZZI
- x = SPAZIO E MOBILITA'
- k = SICUREZZA DEL RE
- c = CONTROLLO DEL CENTRO
- p = STRUTTURA PEDONALE
- a = POSSIBILITA' DI ATTACCO
- r = RELAZIONE TRA I PEZZI

Per riconoscere univocamente una tra le 2^8 possibili funzioni di valutazione (tante quante sono le diverse combinazioni delle 8 porzioni di conoscenza), è sufficiente specificare quali tra le 8 euristiche sono presenti in essa; ad esempio, con "mka" si identifica la funzione che valuta il materiale, la sicurezza del re e le possibilità di attacco.
Una facile codifica delle funzioni di valutazione così identificate si ottiene per mezzo di 8 bit, associando ognuno di essi ad una porzione di conoscenza: il bit si pone a 1 se la sua categoria è da tenere in considerazione, e si pone a 0 altrimenti. Si ricava in questo modo un intero appartenente all'intervallo $[0, 2^8 - 1]$.
La semplice aggiunta nel codice della funzione di valutazione di Gnuchess 4.0 di controlli dinamici da verificare rispetto al numero che identifica la funzione desiderata consente così di variare il flusso di controllo, facendo eseguire ogni volta solamente le routine di valutazione legate alle porzioni di conoscenza da considerare.
Si è ottenuta in questo modo una funzione di valutazione parametrica rispetto al numero che identifica le porzioni di conoscenza associate ad essa.

Siccome la valutazione del materiale (m) è l'euristica di gran lunga più importante, senza la quale non risulterebbe molto affidabile la valutazione di una posizione, allora la sua gestione è stata definita diversamente da quella delle altre euristiche. In pratica, le routine che valutano il materiale sono sempre eseguite, e forniscono sempre il valore previsto, indipendentemente dalla conoscenza associata all'istanza; questo valore viene sommato per intero nel caso che l'euristica m sia compresa nella funzione di valutazione, mentre viene prima diviso per 5 in caso contrario. In questo modo, ad esempio, una funzione di valutazione che non preveda l'esame del materiale deve comunque compensare ogni pedone di svantaggio con un vantaggio di 20 punti ottenuti rispetto ad un'altra euristica.

Capitolo 6

Sperimentazione di Gnupar

L'obiettivo che ci si propone in questo capitolo è quello di valutare i giocatori artificiali ottenuti con l'approccio della distribuzione della conoscenza, cercando di misurarne la forza non più rispetto al comportamento di fronte a singole posizioni, ma nella disputa di partite complete. La loro sperimentazione verrà eseguita su un'architettura parallela composta da una rete locale di 11 SUN SparcStation collegate in Ethernet.
Si proverà inoltre a migliorare le prestazioni dei giocatori paralleli intervenendo nella regolazione di alcuni criteri di selezione.

6.1 Definizione della sperimentazione

Il modo migliore per giudicare un giocatore di scacchi consiste nell'osservarlo alle prese con una partita completa. Utilizziamo a questo proposito l'esperienza di Tozzi [29], che qui riassumiamo brevemente.
Innanzitutto, è necessario definire degli esperimenti di durata non eccessiva e statisticamente attendibili. La soluzione più corretta sarebbe quella di far disputare un torneo di N partite fra ciascuna delle possibili coppie di giocatori paralleli, ma ciò è praticamente improponibile, dato il numero elevato di giocatori che scaturiscono dalla diversa distribuzione delle euristiche e dal grande numero di criteri di selezione utilizzabili. Per questo motivo si seleziona un ristretto numero di giocatori, e li si fa giocare contro un avversario unico; assegnando ai giocatori un punteggio dipendente dal risultato del torneo, è possibile confrontare indirettamente tra loro i giocatori esaminati.
L'avversario unico scelto da Tozzi è stata la versione originale che egli ha usato per la realizzazione dei giocatori paralleli, cioè Gnuchess 3.1. Questa soluzione ha il pregio di verificare direttamente i benefici ottenuti nel distribuire la conoscenza, mettendo a confronto ogni giocatore parallelo con il giocatore sequenziale in cui è concentrata tutta la conoscenza.
Il punteggio assegnato ai giocatori consiste nella sommatoria dei punti ottenuti in ogni singola partita, come avviene in un normale torneo. I punti assegnati in una partita risentono di una particolare condizione imposta per

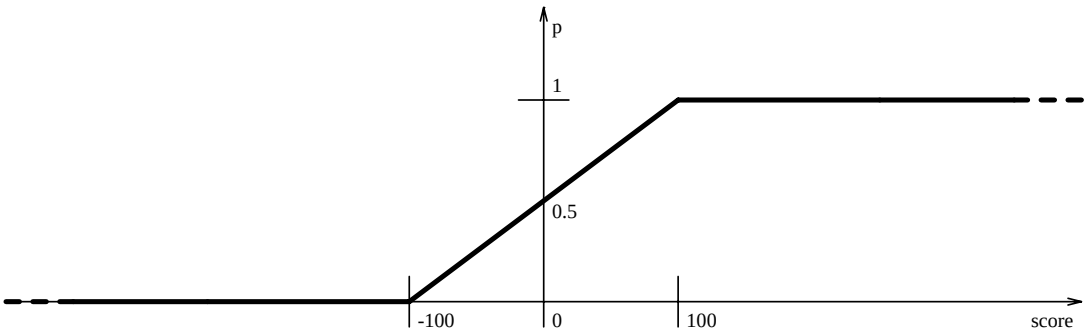

Figura 6.1: Punteggio assegnato ad una partita interrotta secondo Tozzi

limitare sia l'ampiezza della valutazione (confinata per quanto possibile nel mediogioco) che il tempo necessario alla sperimentazione: ogni partita infatti viene troncata dopo un prefissato numero di mosse.
Una partita può dunque terminare o perché è stata raggiunta una posizione finale (patta o matto), o perché è stato raggiunto il limite massimo di mosse.
La definizione dei punti p da assegnare ad un giocatore per ogni partita disputata è secondo Tozzi la seguente (vedere figura 6.1):

- $p = 0$ se la partita è terminata con la sconfitta o è stata interrotta con $score < -PV$;
- $p = 1$ se la partita è terminata con la vittoria o è stata interrotta con $score > PV$;
- $p = 0.5$ se la partita è terminata con la patta;
- $p = \frac{score+PV}{2*PV}$ se la partita è stata interrotta con $-PV \leq score \leq PV$;

dove PV è il valore materiale di un pedone (100), mentre $score$ è la stima del valore della posizione finale, eseguita per mezzo della funzione di valutazione statica di Gnuchess 3.1.
La sperimentazione di Tozzi ha avuto luogo alle seguenti condizioni:

1. sono state esaminate 16 combinazioni di istanze e 2 criteri di selezione, per un totale di 32 giocatori paralleli;
2. ogni giocatore parallelo ha disputato contro Gnuchess 3.1 un torneo di 20 partite, metà col bianco e metà col nero, con un tempo per mossa di 30 secondi. Le partite non sono state eseguite in ambiente distribuito, ma simulate attraverso codice scritto completamente in C;
3. è stato inibito sia per Gnuchess 3.1 che per il suo avversario l'uso del libro delle aperture e della tabella delle trasposizioni; inoltre per entrambi sono state disabilitate le funzionalità di amministrazione del tempo straordinario (ExtraTime) e di ricerca durante l'attesa della mossa dell'avversario.

La sperimentazione è stata preceduta da una serie di partite test tra Gnuchess 3.1 e se stesso (senza il libro delle aperture), volte a valutare quanto questo programma favorisca uno dei due colori: il risultato è stato del 49% di vittorie del bianco, del 33% di vittorie del nero e del 18% di pareggi. Questi valori non differiscono molto da quelli che si riscontrano nella pratica agonistica dei giocatori umani, quindi Gnuchess 3.1 è stato considerato un avversario valido per la valutazione sperimentale dei giocatori paralleli.

6.2 Verifica iniziale

Innanzitutto si è ritenuto di dovere verificare il software a nostra disposizione rispetto ai risultati ottenuti da Tozzi, ripetendo i tornei di due dei suoi migliori giocatori paralleli.
Abbiamo sottoposto Gnuchess 4.0 al test per accertarne la "taratura", facendolo giocare contro se stesso (concedendogli in questo caso la possibilità di usare il libro delle aperture): dopo 100 partite è stato sostanzialmente confermato il risultato di Tozzi, avendo ottenuto 46 vittorie del bianco, 36 del nero e 18 pareggi. Anche Gnuchess 4.0 può quindi essere considerato un avversario statisticamente valido.
I due giocatori paralleli considerati sono quelli composti da 6 istanze a distribuzione minima (mb mx mk mp ma mr) e bilanciata (mbxk mbka mbpr mxpa mxar mkpr), con il criterio di selezione a maggioranza generalizzata con pesi costanti, determinati assegnando alle euristiche i pesi m $\Rightarrow$ 30, b $\Rightarrow$ 21, x $\Rightarrow$ 15, k $\Rightarrow$ 10, p $\Rightarrow$ 6, a $\Rightarrow$ 3 ed r $\Rightarrow$ 1. A tale proposito si fa notare che il peso delle istanze era stabilito nei giocatori di Tozzi calcolando la media dei pesi delle euristiche associate, diversamente da quanto avviene in Gnupar.
Questi due giocatori avevano ottenuto un punteggio rispettivamente di 13.53 (con 6 vittorie col bianco e 6 col nero) e di 14.05 (con 9 vittorie col bianco e 4 col nero), battendo nettamente l'avversario comune Gnuchess 3.1.
Per la ripetizione dell'esperimento sono stati definiti due appositi giocatori paralleli, ricavati da Gnuchess 4.0 con la stessa suddivisione delle euristiche all'interno del codice usata da Tozzi (m = materiale, b = valore posizionale dei pezzi, x = mobilità di torre ed alfiere e combinazioni di attacco, k = sicurezza del re, a = protezione dei pezzi, r = relazione pezzi-struttura pedonale), e con lo stesso metodo di determinazione del peso delle istanze. Diversamente dall'esperimento di Tozzi, i tornei sono stati disputati in ambiente distribuito; sono stati inoltre usati sia il libro delle aperture che la tabella delle trasposizioni, ed è stato definito un nuovo punteggio da assegnare al giocatore in funzione dell'esito di una partita, in quanto si ritiene che, nel caso di interruzione per raggiunto limite di mosse, quello usato negli esperimenti di Tozzi premiasse eccessivamente un vantaggio piccolo come quello equivalente al guadagno di un pedone. La nuova definizione dei punti p da assegnare ad un giocatore per ogni partita disputata è la seguente (vedere figura 6.2):

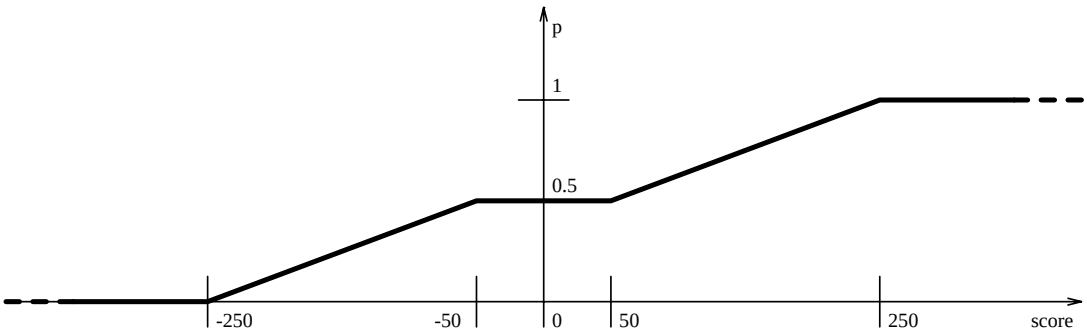

Figura 6.2: Nuovo punteggio assegnato ad una partita interrotta

- $p = 0$ se la partita è terminata con la sconfitta o è stata interrotta con $score < -250$;
- $p = 1$ se la partita è terminata con la vittoria o è stata interrotta con $score > 250$;
- $p = 0.5$ se la partita è terminata con la patta o è stata interrotta con $|score| < 50$;
- $p = \frac{score+150}{400}$ se la partita è stata interrotta con $50 \leq score \leq 250$;
- $p = 1 - \frac{|score|+150}{400}$ se la partita è stata interrotta con $-250 \leq score \leq -50$;

dove *score* è la stima del valore della posizione finale, eseguita per mezzo della funzione di valutazione statica di Gnuchess 4.0.

I risultati dei due tornei sono indicati nelle tabelle che seguono. Notare che in esse sono distinte le valutazioni delle partite interrotte nel momento in cui è stata eseguita la cattura di un pezzo (posizioni instabili, indicate con "c") dalle altre terminate normalmente.

```
*************************
Gnupar = mb mx mk mp ma mr
*************************
```

		Gnuchess		Gnupar	
part	capt	punti	col	punti	col
1		1.00	W	0.00	B
2		0.35	B	0.65	W
3		1.00	W	0.00	B
4		1.00	B	0.00	W
5		1.00	W	0.00	B
6		1.00	B	0.00	W
7		1.00	W	0.00	B
8		0.75	B	0.25	W
9		0.75	W	0.25	B
10		1.00	B	0.00	W
11		0.50	W	0.50	B
12		0.00	B	1.00	W
13		0.67	W	0.33	B
14		0.16	B	0.84	W
15		0.00	W	1.00	B
16		0.00	B	1.00	W
17		1.00	W	0.00	B
18		1.00	B	0.00	W
19		1.00	W	0.00	B
20		0.50	B	0.50	W
TOT NORM		13.68		6.32	
TOT CAPT		0.00		0.00	

```
************************************
Gnupar = mbxk mbka mbpr mxpa mxar mkpr
************************************
```

		Gnuchess		Gnupar	
part	capt	punti	col	punti	col
1		0.87	W	0.13	B
2	c	1.00	B	0.00	W
3		0.50	W	0.50	B
4		0.50	B	0.50	W
5	c	0.35	W	0.65	B
6		0.00	B	1.00	W
7	c	0.00	W	1.00	B
8		0.78	B	0.22	W
9		0.50	W	0.50	B
10		0.56	B	0.44	W
11		1.00	W	0.00	B
12		0.50	B	0.50	W
13		0.78	W	0.22	B
14		1.00	B	0.00	W
15		1.00	W	0.00	B
16		1.00	B	0.00	W
17		0.00	W	1.00	B
18		1.00	B	0.00	W
19		1.00	W	0.00	B
20		0.28	B	0.72	W
TOT NORM		11.26		5.74	
TOT CAPT		1.35		1.65	

Evidentemente, l'esito della verifica è negativo: è troppo grande il divario esistente rispetto ai risultati di Tozzi.
Forse ciò è dovuto alle diverse condizioni in cui si sono svolti i tornei, per cui si è ritenuto opportuno ripetere lo stesso esperimento cercando di ridurre al minimo le differenze: si sono disputati gli stessi tornei escludendo questa volta l'uso del libro delle aperture e della tabella delle trasposizioni.
Gli esiti sono riportati nelle tabelle che seguono.

```
*************************
Gnupar = mb mx mk mp ma mr
*************************
```

		Gnuchess		Gnupar	
part	capt	punti	col	punti	col
1		0.58	W	0.42	B
2		1.00	B	0.00	W
3		0.12	W	0.88	B
4	c	0.80	B	0.20	W
5		1.00	W	0.00	B
6		0.74	B	0.26	W
7		0.27	W	0.73	B
8		1.00	B	0.00	W
9	c	1.00	W	0.00	B
10	c	0.00	B	1.00	W
11		0.50	W	0.50	B
12		1.00	B	0.00	W
13		0.50	W	0.50	B
14	c	0.98	B	0.02	W
15		1.00	W	0.00	B
16		0.00	B	1.00	W
17		1.00	W	0.00	B
18		1.00	B	0.00	W
19		1.00	W	0.00	B
20		1.00	B	0.00	W
TOT NORM		11.72		4.28	
TOT CAPT		2.78		1.22	

```
************************************
Gnupar = mbxk mbka mbpr mxpa mxar mkpr
************************************
```

		Gnuchess		Gnupar	
part	capt	punti	col	punti	col
1		1.00	W	0.00	B
2		0.16	B	0.84	W
3		1.00	W	0.00	B
4		0.50	B	0.50	W
5	c	0.39	W	0.61	B
6		0.52	B	0.48	W
7		0.00	W	1.00	B
8		0.50	B	0.50	W
9	c	0.96	W	0.04	B
10		0.44	B	0.56	W
11		0.46	W	0.54	B
12		0.63	B	0.37	W
13		1.00	W	0.00	B
14		1.00	B	0.00	W
15		0.00	W	1.00	B
16		1.00	B	0.00	W
17		0.50	W	0.50	B
18		0.50	B	0.50	W
19		0.00	W	1.00	B
20		1.00	B	0.00	W
TOT NORM		10.21		7.79	
TOT CAPT		1.36		0.64	

Alla luce di questi risultati si deve dunque escludere un qualsiasi confronto tra i valori ricavati per i nostri giocatori paralleli ed i valori che erano stati determinati nell'esperimento di Tozzi. Rimane comunque valido il metodo per la comparazione delle prestazioni dei vari giocatori paralleli, costituito dalla disputa di tornei contro un avversario comune.

6.3 Esecuzione dei tornei

Le funzioni adottate sia nell'esperimento di Tozzi che nella verifica precedente, usate per definire i punti da assegnare ad un giocatore per ogni partita disputata, utilizzavano nel caso di interruzione per raggiunto limite massimo di mosse la funzione di valutazione statica di Gnuchess 4.0, e ne ereditavano quindi tutti i relativi punti deboli. In particolare, esse lasciavano il dubbio sull'effettivo valore delle posizioni instabili, tanto da richiedere una loro classificazione separata. Per questo motivo si è ritenuto di dovere determinare la stima del valore della posizione in cui è avvenuta l'interruzione (*score*) applicando una valutazione dinamica, facendo cioè eseguire per un tempo prestabilito (60 secondi) e con la totale conoscenza di Gnuchess 4.0 un'ulteriore ricerca da parte del colore a cui tocca muovere. I punti di ogni partita saranno determinati d'ora in avanti utilizzando la stessa funzione di pag. 90, ma considerando il valore di *score* calcolato come sopra.
Esaminiamo a questo punto le prestazioni di Gnupar, studiando le distribuzioni della conoscenza che hanno dimostrato di avere le migliori potenzialità, cioè quelle a 7 istanze con distribuzione bilanciata (mpar mbxp mbkc mxar mkcp mcar mbxk) e sbilanciata (mbxkcpar mb ma mc mbkcpar mbxcpar mbxkcpa), ed i criteri di selezione che hanno raggiunto con continuità le migliori prestazioni, cioè i criteri "pesi", "depth" e "re_search".

6.3.1 Criterio "pesi"

Utilizzando il criterio "pesi" così come definito a pag. 67, è stato eseguito un torneo per entrambi i giocatori (quello a distribuzione bilanciata e quello a distribuzione sbilanciata) assegnando per ogni mossa un tempo di 30 secondi. I risultati sono mostrati nelle tabelle che seguono.

```
*****************************************
Gnupar = mpar mbxp mbkc mxar mkcp mcar mbxk
*****************************************
```

	Gnuchess		Gnupar	
part	punti	col	punti	col
1	1.00	W	0.00	B
2	0.23	B	0.77	W
3	0.60	W	0.40	B
4	0.50	B	0.50	W
5	0.40	W	0.60	B
6	1.00	B	0.00	W
7	1.00	W	0.00	B
8	0.00	B	1.00	W
9	0.00	W	1.00	B
10	0.00	B	1.00	W
11	0.00	W	1.00	B
12	0.00	B	1.00	W
13	0.00	W	1.00	B
14	1.00	B	0.00	W
15	0.70	W	0.30	B
16	1.00	B	0.00	W
17	1.00	W	0.00	B
18	1.00	B	0.00	W
19	0.00	W	1.00	B
20	0.00	B	1.00	W
TOT	9.44		10.56	

```
************************************************
Gnupar = mbxkcpar mb ma mc mbkcpar mbxcpar mbxkcpa
************************************************
```

	Gnuchess			Gnupar		
part	punti	col		punti	col	
1	1.00	W		0.00	B	
2	0.00	B		1.00	W	
3	0.50	W		0.50	B	
4	1.00	B		0.00	W	
5	0.00	W		1.00	B	
6	0.43	B		0.57	W	
7	0.00	W		1.00	B	
8	0.00	B		1.00	W	
9	1.00	W		0.00	B	
10	0.16	B		0.84	W	
11	0.63	W		0.37	B	
12	0.50	B		0.50	W	
13	0.00	W		1.00	B	
14	1.00	B		0.00	W	
15	1.00	W		0.00	B	
16	0.47	B		0.53	W	
17	1.00	W		0.00	B	
18	0.00	B		1.00	W	
19	0.50	W		0.50	B	
20	0.00	B		1.00	W	
TOT	9.20			10.80		

6.3.2 Criterio "depth"

Utilizzando il criterio "depth" così come definito a pag. 67, è stato eseguito un torneo con il giocatore a distribuzione bilanciata, assegnando per ogni mossa un tempo di 30 secondi.
I risultati sono mostrati nella tabella che segue.

```
************************************
Gnupar = mbxk mbka mbpr mxpa mxar mkpr
************************************
```

	Gnuchess			Gnupar		
part	punti	col		punti	col	
1	0.00	W		1.00	B	
2	1.00	B		0.00	W	
3	0.00	W		1.00	B	
4	0.69	B		0.31	W	
5	0.81	W		0.19	B	
6	1.00	B		0.00	W	
7	0.00	W		1.00	B	
8	0.00	B		1.00	W	
9	1.00	W		0.00	B	
10	0.50	B		0.50	W	
11	0.96	W		0.04	B	
12	1.00	B		0.00	W	
13	1.00	W		0.00	B	
14	0.00	B		1.00	W	
15	1.00	W		0.00	B	
16	1.00	B		0.00	W	
17	0.00	W		1.00	B	
18	0.00	B		1.00	W	
19	0.23	W		0.77	B	
20	0.50	B		0.50	W	
TOT	10.70			9.30		

6.3.3 Criterio "re_search"

In accordo con la definizione del criterio eseguita a pag. 68, si è assegnato lo stesso tempo alla selezione parallela ed alla ricerca successiva (30 secondi). Si fa osservare che, quando tutte le istanze scelgono la stessa mossa, non viene eseguita la ricerca successiva. Questa può essere considerata una soluzione al problema delle mosse che non portano allo scacco matto ma che sono evidentemente obbligate: generalmente, un giocatore artificiale impiega anche in tal caso tutto il suo tempo di elaborazione, diversamente dalla presente realizzazione di Gnupar.
È stato quindi eseguito un torneo con il giocatore a distribuzione bilanciata

assegnando per ogni mossa un tempo di 60 secondi. I risultati sono mostrati nella tabella che segue.

```
************************************
Gnupar = mbxk mbka mbpr mxpa mxar mkpr
************************************
```

	Gnuchess			Gnupar		
part	punti	col		punti	col	
1	0.50	W		0.50	B	
2	0.50	B		0.50	W	
3	1.00	W		0.00	B	
4	0.50	B		0.50	W	
5	1.00	W		0.00	B	
6	1.00	B		0.00	W	
7	1.00	W		0.00	B	
8	0.00	B		1.00	W	
9	1.00	W		0.00	B	
10	0.00	B		1.00	W	
11	0.50	W		0.50	B	
12	1.00	B		0.00	W	
13	0.00	W		1.00	B	
14	1.00	B		0.00	W	
15	0.39	W		0.61	B	
16	1.00	B		0.00	W	
17	0.00	W		1.00	B	
18	1.00	B		0.00	W	
19	1.00	W		0.00	B	
20	0.00	B		1.00	W	
TOT	12.40			7.60		

6.4 Valutazione dei risultati

I risultati ottenuti non hanno mai favorito, se non in piccola misura, il giocatore parallelo: questo non giustifica certamente il grande dispendio di risorse di elaborazione (con 7 processori si è avuto nel caso migliore un risultato di sostanziale equilibrio nei confronti di un solo processore).
Con opportune variazioni nelle definizioni di alcuni criteri di selezione, cercheremo di determinare giocatori in grado di migliorare le prestazioni attuali.

6.5 Miglioramento del criterio "pesi"

Ad ogni istanza del giocatore parallelo viene attribuito un peso dipendente dalle euristiche considerate nella funzione di valutazione associata: è quindi il peso assegnato ad ogni euristica che determina il valore costante delle istanze. Siccome il peso delle istanze ha un'importanza fondamentale nella decisione della mossa definitiva, risulta evidente che al variare dei pesi associati alle euristiche possono variare le mosse scelte dal giocatore parallelo, e di conseguenza anche le sue prestazioni.
Per il miglioramento del criterio di selezione si tratta dunque di associare alle euristiche quei pesi che consentano di selezionare più spesso una mossa buona tra quelle proposte dalle istanze. Prendendo spunto da un esperimento volto ad ottimizzare i pesi assegnati ai vari termini di conoscenza all'interno di una funzione di valutazione statica [30], cerchiamo di ricavare, utilizzando un algoritmo genetico e le 500 posizioni, una combinazione di pesi che permetta di migliorare le prestazioni di Gnupar.

6.5.1 Algoritmi genetici

Un algoritmo genetico è un procedimento di ottimizzazione: la sua definizione può essere fatta risalire a Bagley [4] e Holland [19], con la successiva rielaborazione di Goldgerg nell'ambito dei metodi di apprendimento automatico [18]. Esso è basato su un insieme di postulati tratti dalla teoria darwiniana della selezione naturale, secondo i quali varietà ed ereditarietà dei caratteri sono condizioni che garantiscono la capacità evolutiva delle popolazioni di esseri viventi. Gli individui con le caratteristiche che meglio si adattano all'ambiente sono anche quelli con le maggiori probabilità di riproduzione; siccome a causa dell'ereditarietà queste caratteristiche vengono trasmesse anche ai figli, si ha in pratica un'evoluzione verso un migliore adattamento all'ambiente. La possibilità del verificarsi di mutazioni consente inoltre una maggiore variabilità della popolazione; da una mutazione potrebbe casualmente derivare un individuo con caratteristiche che porteranno ad un ulteriore miglioramento nell'adattamento.
In un computer, un algoritmo genetico simula il procedimento appena descritto. Dato un problema di cui si vogliono determinare le migliori soluzioni, si definisce una popolazione di possibili soluzioni, e la si fa evolvere verso soluzioni migliori. Ogni individuo della popolazione viene identificato con una sequenza di bit, che idealmente rappresentano i geni del suo corredo cromosomico; la validità della soluzione al problema rappresentata dall'individuo viene misurata per mezzo di un'apposita funzione di adattamento (fitness function) la quale associa ad ogni individuo un valore tanto maggiore quanto migliore è la soluzione rappresentata da esso per il problema in oggetto. Così come l'adattamento di una specie si concretizza con la proliferazione, anche in un algoritmo genetico è previsto l'accoppiamento e la riproduzione degli individui della popolazione. In particolare, si gestisce una successione continua di

generazioni: ad ogni nuovo passaggio, cioè, si sostituisce l'intera popolazione attuale con una popolazione generata propagando con maggiore probabilità i caratteri degli individui più adatti. La propagazione dei caratteri è garantita dal meccanismo di riproduzione, nel quale, analogamente a quanto avviene nella realtà, si realizza uno scambio di porzioni del corredo cromosomico dei genitori a formare il corredo cromosomico dei figli (cross-over). Per completare l'analogia col processo genetico reale, viene introdotta una componente di variabilità consentendo una sporadica mutazione dei geni contenuti nei cromosomi dei figli.
La realizzazione dell'algoritmo genetico per la ottimizzazione dei pesi da dare alle euristiche si basa sulla seguente definizione dei concetti previsti:

1. popolazione: il numero N di individui di cui si vuole osservare l'evoluzione è fisso. Si suppone che la dimensione della popolazione rimanga costante nel tempo, e questo si ottiene facendo generare due figli per ogni coppia di individui.

2. codifica degli individui: si sceglie di limitare tra 0 e 255 il valore che può assumere il peso da associare ad un'euristica, per cui è rappresentabile con 8 bit. Essendovi 8 euristiche, si codifica ogni individuo per mezzo di 64 bit. Ai gruppi consecutivi di 8 bit, da quelli più significativi a quelli meno significativi, sono associati nell'ordine i pesi delle euristiche m b x k c p a r.

3. nel nostro caso, l'individuo più adatto è quello che rappresenta i pesi migliori per il giocatore parallelo considerato, e questo si riconosce contando il numero di mosse buone che tale giocatore parallelo seleziona nell'esame delle 500 posizioni. Per ampliare il divario esistente tra individui che consentono prestazioni differenti, si definisce la seguente fitness function:
$$ff(I) = 1 \quad se\ npos(I) < 70$$
$$ff(I) = \frac{(npos(I) - 70)^3}{500} + 1 \quad altrimenti$$
dove $npos(I)$ è il numero di posizioni in cui il giocatore parallelo coi pesi determinati dall'individuo I seleziona una mossa buona.

4. accoppiamento: maggiore è la capacità di adattamento di un individuo e maggiore deve essere la probabilità che ha di accoppiarsi. Supponendo che ogni individuo si possa accoppiare con più individui ed anche con se stesso, si estraggono casualmente N individui per mezzo del metodo Montecarlo, in modo che la frequenza degli individui estratti sia proporzionale al valore determinato per essi dalla fitness function, dopodiché si accoppiano questi individui a due a due.

5. riproduzione: dati due individui genitori, si ricavano da essi due figli eseguendo in successione le seguenti operazioni:

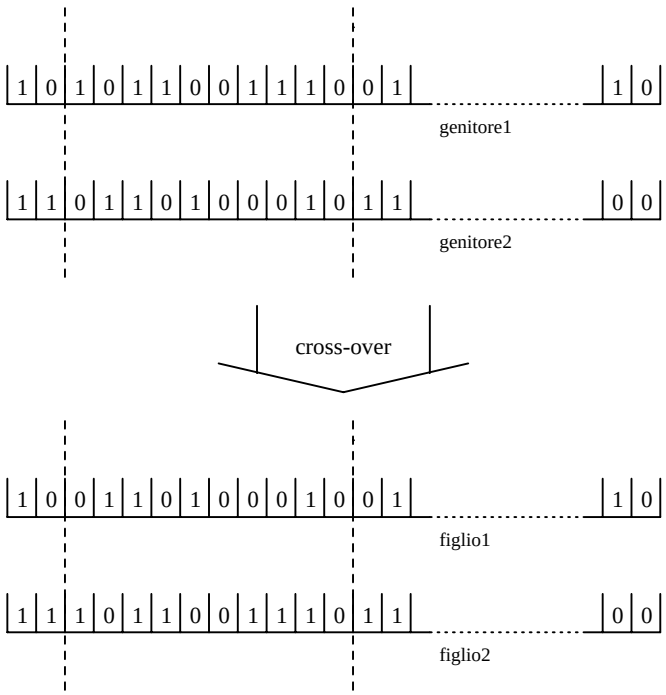

Figura 6.3: Esempio di cross-over

- con probabilità PC si esegue il cross-over dei cromosomi dei genitori, oppure con probabilità $1 - PC$ si generano due figli esattamente uguali ai genitori.
  Il cross-over consiste nell'individuazione casuale di una porzione del corredo cromosomico dei genitori, e nello scambio di questa porzione da un genitore all'altro (vedere figura 6.3);

- si scorrono uno per uno i bit di entrambi i figli, invertendo con probabilità PM il loro valore (mutazione).

Per l'esecuzione dell'esperimento sono stati scelti i parametri $N = 30$ (numero di individui della popolazione), $PC = 0.6$ (probabilità di cross-over), $PM = 0.01$ (probabilità di mutazione).
L'algoritmo non ha una convergenza dimostrata: semplicemente, con buona probabilità fa tendere gli individui a valori sempre migliori rispetto al problema in esame. Il metodo da noi usato per cercare di determinare pesi migliori da associare alle euristiche consiste nell'eseguire a più riprese un certo numero di generazioni successive: quando si sono determinati pesi che permettono al giocatore parallelo di selezionare un numero consistente di mosse buone nella valutazione delle 500 posizioni, allora si può interrompere l'algoritmo e passare alla sperimentazione tramite torneo.

6.5.2 Prova del miglioramento

Dopo aver eseguito per il giocatore con 7 istanze a distribuzione bilanciata l'algoritmo genetico descritto fino a portare il numero di posizioni in cui è stata selezionata una mossa buona da 88 (cfr cap. 4.3.2) a 104, sono stati ricavati i pesi m $\Rightarrow$ 45, b $\Rightarrow$ 24, x $\Rightarrow$ 254, k $\Rightarrow$ 194, c $\Rightarrow$ 38, p $\Rightarrow$ 223, a $\Rightarrow$ 15 ed r $\Rightarrow$16. È stato quindi disputato un torneo assegnando questi pesi alle euristiche, ottenendo i seguenti risultati.

```
*****************************************
Gnupar = mpar mbxp mbkc mxar mkcp mcar mbxk
*****************************************
```

	Gnuchess			Gnupar		
part	punti	col		punti	col	
1	0.39	W		0.61	B	
2	0.50	B		0.50	W	
3	0.00	W		1.00	B	
4	1.00	B		0.00	W	
5	1.00	W		0.00	B	
6	1.00	B		0.00	W	
7	0.00	W		1.00	B	
8	0.00	B		1.00	W	
9	0.00	W		1.00	B	
10	0.00	B		1.00	W	
11	0.00	W		1.00	B	
12	0.33	B		0.67	W	
13	0.50	W		0.50	B	
14	0.00	B		1.00	W	
15	0.00	W		1.00	B	
16	0.00	B		1.00	W	
17	1.00	W		0.00	B	
18	0.28	B		0.72	W	
19	0.00	W		1.00	B	
20	0.00	B		1.00	W	
TOT	5.99			14.01		

6.6 Miglioramento del criterio "re_search"

Ciò che può variare nella definizione di un giocatore parallelo con criterio di selezione "re_search" è la suddivisione del tempo per mossa tra la selezione parallela e la ricerca successiva. Supponendo intuitivamente che sia meglio utilizzare la maggior parte del tempo per la ricerca finale, in quanto un minore approfondimento delle mosse proposte dalle istanze dovrebbe essere sufficientemente compensato dal loro numero, abbiamo assegnato 1/3 del tempo alla selezione parallela e 2/3 alla ricerca successiva. È stato quindi disputato dal giocatore con 7 istanze a distribuzione bilanciata un torneo con 60 secondi per mossa, i cui risultati sono mostrati nella tabella che segue.

```
*****************************************
Gnupar = mpar mbxp mbkc mxar mkcp mcar mbxk
*****************************************
```

	Gnuchess		Gnupar	
part	punti	col	punti	col
1	0.48	B	0.52	W
2	0.00	W	1.00	B
3	0.00	B	1.00	W
4	1.00	W	0.00	B
5	0.00	B	1.00	W
6	0.87	W	0.13	B
7	0.74	B	0.26	W
8	0.50	W	0.50	B
9	0.10	B	0.90	W
10	0.50	W	0.50	B
11	0.62	B	0.38	W
12	1.00	W	0.00	B
13	0.00	B	1.00	W
14	0.00	W	1.00	B
15	0.00	B	1.00	W
16	0.00	W	1.00	B
17	0.50	B	0.50	W
18	0.50	W	0.50	B
19	0.00	B	1.00	W
20	0.00	W	1.00	B
TOT	6.82		13.18	

Visto il buon risultato conseguito, abbiamo ampliato la sperimentazione di questo giocatore parallelo facendolo giocare, per mezzo dell'interfaccia Xboard, anche con giocatori umani, sia direttamente alla console che tramite l'ICS: alcune partite esemplificative sono mostrate in appendice B.

Capitolo 7

Conclusioni

In questo lavoro è stato preso in esame un approccio relativamente nuovo al progetto di giocatori artificiali paralleli, denominato distribuzione della conoscenza. L'idea alla base di tale metodo di parallelizzazione è la cooperazione tra giocatori sequenziali con diversa conoscenza del dominio di applicazione: ogni giocatore propone la sua migliore mossa e, dall'insieme così ottenuto di mosse candidate, viene scelta quella che sarà effettivamente giocata.
Allo scopo di verificare le qualità di tale approccio, abbiamo esaminato la sua applicazione in un particolare campo d'interesse: gli scacchi.

7.1 Lavoro svolto

Attraverso il riuso di software tratto da Gnuchess 4.0 (un programma sequenziale di scacchi di dominio pubblico), e grazie al linguaggio di coordinazione Network C-Linda (basato sul modello a spazio di tuple per la creazione e la coordinazione di processi), abbiamo prodotto un giocatore artificiale di scacchi che realizza la distribuzione della conoscenza, denominato Gnupar.
Tale giocatore artificiale è risultato molto versatile, in quanto può distribuire su un numero variabile di processori combinazioni differenti dei termini di conoscenza tratti dalla funzione di valutazione di Gnuchess 4.0, ed inoltre può eseguire la scelta della mossa finale secondo diverse modalità. Dato il grande numero di giocatori che si possono ottenere con Gnupar, abbiamo eseguito alcuni esperimenti per determinare da quali distribuzioni della conoscenza e da quali criteri di selezione della mossa finale scaturiscano i giocatori potenzialmente migliori. Innanzitutto si è provveduto ad ordinare i termini di conoscenza di Gnuchess 4.0 secondo il miglioramento delle prestazioni indotto dalla loro introduzione nella funzione di valutazione di un giocatore artificiale.
Successivamente si sono individuati alcuni criteri di distribuzione della conoscenza dipendenti dall'ordinamento appena descritto: da tali criteri è stato tratto un certo numero di distribuzioni effettive, di cui si sono stimate speri-

mentalmente le potenzialità.
Si è infine valutata la validità di alcuni criteri di selezione applicati alle precedenti distribuzioni effettive.
Gli esperimenti descritti sono consistiti nell'esplorazione di un notevole numero (500) di posizioni scacchistiche da parte di giocatori artificiali sequenziali con conoscenza ridotta, rappresentanti le istanze del giocatore parallelo da determinare.
Successivamente si è passati alla sperimentazione effettiva di Gnupar. La prima operazione svolta è stata una verifica della corrispondenza tra il comportamento di Gnupar e quello dell'analogo giocatore artificiale con distribuzione della conoscenza esaminato da Tozzi in [29]. La verifica ha avuto esito negativo: gli esperimenti eseguiti hanno dimostrato che i due giocatori, anche se posti nelle stesse condizioni , si comportano in maniera diversa. Questo fatto ci ha portato ad escludere un qualsiasi confronto tra i risultati ottenuti da Gnupar e quelli ottenuti dal giocatore artificiale di Tozzi. Ciò non ha escluso comunque l'uso dello stesso metodo di valutazione suggerito da Tozzi: la disputa di tornei di 20 partite tra il giocatore parallelo da analizzare e la versione da cui è stato tratto.
Si è quindi provveduto all'applicazione dei criteri di distribuzione e di selezione emersi dalle stime precedenti su un certo numero di giocatori, ed alla sperimentazione di questi tramite tornei contro 4.0. Gli esiti dei tornei non hanno mai favorito (se non in piccola misura) il giocatore parallelo, cosa che non giustifica sicuramente il maggiore impiego di risorse di elaborazione. Si è quindi pensato di apportare alcune modifiche alla regolazione di alcuni criteri di selezione, per tentare di migliorare le prestazioni.
Nel criterio "pesi" (la cui definizione è a pag. 67), grande importanza ha la determinazione del peso da associare alle istanze che propongono le mosse, ricavato dalla sommatoria dei pesi delle porzioni di conoscenza considerate nella valutazione: attraverso un algoritmo genetico è stata trovata una combinazione di pesi da associare ai termini di conoscenza, che ha apportato un netto miglioramento alle prestazioni del giocatore parallelo corrispondente.
Nel criterio "re_search" (definito a pag. 68), invece, risulta importante la suddivisione, tra la selezione parallela e la ricerca successiva, del tempo a disposizione per eseguire la mossa: definendo intuitivamente un rapporto diverso da quello originario (1/3 e 2/3 invece che 1/2 e 1/2), si è rilevato anche in questo caso un notevole miglioramento nelle prestazioni del giocatore parallelo derivato.
In conclusione, tramite Gnupar è stato dimostrato che la distribuzione della conoscenza può determinare un consistente miglioramento nelle prestazioni di un giocatore artificiale. Esistono infatti almeno due versioni di Gnupar che battono nettamente il programma sequenziale originale, e, data l'estrema aleatorietà di alcune scelte eseguite per arrivare a questi risultati, non è escluso che esistano versioni ancora migliori.
Per quanto riguarda l'uso del linguaggio di coordinazione Network C-Linda,

si deve rilevare che esso si è dimostrato facilmente integrabile con software esistente; inoltre, il programma risultante si è rivelato di notevole affidabilità: durante la sperimentazione, infatti, non sono mai stati riscontrati particolari problemi, nonostante l'esecuzione di un considerevole numero di partite e le difficoltà proposte dall'interfacciamento con Xboard.

7.2 Prospettive future

Un aspetto tipico dei lavori riguardanti i programmi di scacchi è la rilevante componente sperimentale: ad esempio, per avere dati sufficienti ad esprimere un giudizio sull'efficacia di una nuova idea, può essere necessario eseguire la valutazione di centinaia o migliaia di posizioni. La natura probabilistica della visita degli alberi di gioco assegna alla sperimentazione un ruolo consistente nell'analisi degli algoritmi di ricerca.
Senza sottovalutare l'importanza della sperimentazione nello sviluppo dei programmi di scacchi, risulta evidente che ci si affida spesso troppo pesantemente ad essa senza necessariamente comprendere la teoria sottostante. In pratica, solo dopo aver ottenuto i risultati si cerca di capire cos'è successo, e questo contrasta nettamente con quanto avviene in altre discipline sperimentali: nella fisica, per esempio, gli esperimenti sono eseguiti generalmente per confermare risultati teorici, e raramente avviene il contrario.
Il problema principale è la mancanza di una teoria sui programmi di scacchi: esistono sì degli approfondimenti teorici sugli algoritmi di ricerca (sull'Alpha-Beta in particolare), ma non c'è ancora una teoria sulla conoscenza specifica e sulle sue interazioni con la ricerca. Si verifica così che i programmi vengono modificati senza avere la possibilità di comprendere le potenziali conseguenze del cambiamento, nella speranza di ottenere dei risultati positivi.
Anche nel nostro caso, nonostante l'applicazione di un metodo di lavoro per quanto possibile rigoroso e sistematico, ci si è affidati in molte circostanze a decisioni intuitive: la scomposizione della funzione di valutazione di Gnuchess 4.0 in porzioni di conoscenza, la definizione dei criteri di selezione, la distribuzione della conoscenza tra le istanze del giocatore parallelo, ecc. È chiaro che in mancanza di una teoria al riguardo non si può che proporre la sperimentazione di scelte diverse da quelle eseguite, nella speranza di ottenere prestazioni ancora migliori.
Presa per buona la definizione attuale di Gnupar, risulta interessante l'analisi delle motivazioni che hanno portato al miglioramento delle prestazioni di alcuni giocatori, ed in particolare:

- criterio "pesi": che rapporto c'è tra i pesi associati alle porzioni di conoscenza dall'algoritmo genetico, e l'importanza che queste porzioni hanno all'interno della funzione di valutazione?

- criterio "re_search": perché risulta privilegiata una ridotta selezione parallela, e qual è la proporzione ideale di tempo da assegnare?

Un altro interessante argomento d'indagine è la definizione di apposite euristiche che prevedano la comunicazione tra le istanze durante la selezione parallela: la valutazione eseguita da un'istanza con una certa conoscenza può infatti servire ad un'altra istanza con una conoscenza diversa per eseguire eventuali tagli nella valutazione dell'albero di gioco locale. I benefici di tali euristiche sono comunque fortemente influenzati dall'overhead di comunicazione che ne deriverebbe, e devono quindi essere opportunamente studiati.
Un'euristica, già prevista in Gnuchess 4.0, che potrebbe essere introdotta anche nel giocatore parallelo con promettenti prospettive è quella che fa anticipare la visita dell'albero di gioco, ipotizzando che l'avversario giochi la mossa prevista come migliore risposta nella ricerca precedente: se l'avversario eseguirà veramente quella mossa, si avrà un guadagno di tempo di elaborazione, altrimenti la ricerca ricomincerà da capo, come sarebbe avvenuto normalmente.
Infine, dopo aver sottolineato che sarebbe evidentemente interessante confrontare direttamente due giocatori artificiali che realizzano con lo stesso numero di processori l'uno la parallelizzazione della ricerca e l'altro la distribuzione della conoscenza, si propone come ulteriore argomento di ricerca l'integrazione tra le due forme di parallelismo. Può essere per esempio realizzato un giocatore artificiale a distribuzione della conoscenza con criterio di selezione "re_search" in cui la fase di visita successiva dell'albero di gioco viene realizzata per mezzo della parallelizzazione della ricerca. Oppure un giocatore artificiale a distribuzione della conoscenza con criterio di selezione "pesi", in cui ogni istanza esegue la visita dell'albero di gioco locale per mezzo della parallelizzazione della ricerca.
In conclusione, la distribuzione della conoscenza sembra essere un promettente approccio alla parallelizzazione di un giocatore artificiale, e merita quindi di essere tenuta in considerazione per eventuali lavori futuri.

Capitolo 8

Appendice

8.1 Appendice A: comandi di Gnuchess 4.0

I comandi previsti dall'interfaccia utente di Gnuchess 4.0 sono i seguenti:

- modifica dei parametri di ricerca:
  - Awindow: cambia il valore di ϵ per la determinazione iniziale di α nell'aspiration-window;
  - Bwindow: cambia il valore di ϵ per la determinazione iniziale di β nell'aspiration-window;
  - xwndw: cambia la dimensione della finestra attorno ad $\alpha\beta$, usata per determinare se la posizione deve essere valutata staticamente o se è sufficiente una stima approssimativa del suo valore;
  - contempt: permette di modificare il valore assegnato ad una posizione terminale di patta (per ripetizione di mosse, per superamento del limite di 50 mosse o per stallo). Normalmente il valore è 0, ma può essere cambiato per incentivare o per scoraggiare il raggiungimento di tali posizioni;
  - depth: permette di cambiare la massima profondità di ricerca del programma;
  - hashdepth: permette all'utente di cambiare i parametri che limitano l' accesso all'hash file durante la ricerca nell'albero di gioco;
  - level: permette all'utente di ridefinire il tipo di controllo del tempo;
  - time: permette all'utente di modificare il tempo rimanente al computer prima del controllo;
  - otim: permette all'utente di modificare il tempo rimanente all'avversario del computer prima del controllo;
- visualizzazione:
  - bd: visualizza la posizione corrente;

  - coords: mostra le coordinate della scacchiera;
  - help: visualizza una breve descrizione dei comandi e lo stato corrente dei flag su cui agiscono;
  - hint: visualizza come suggerimento la risposta che il computer ha previsto di ricevere nella variante principale;
  - post: fa vedere, durante la ricerca nell'albero, la variante principale in esame ed il suo punteggio;
  - reverse: inverte la visualizzazione della scacchiera;
  - stars: aggiunge un asterisco ai pezzi neri, per meglio distinguerli nella visualizzazione della posizione;

- modifica dei flag:
  - beep: attiva/disattiva il segnale sonoro dopo ogni mossa;
  - book: attiva/disattiva l'uso del libro delle aperture;
  - both: se attivato fa gestire al computer entrambi i colori (computer vs computer);
  - easy: attiva/disattiva la possibilità di far pensare il computer durante il tempo dell'avversario;
  - gamein: attiva/disattiva il modo partita, il quale assume che il tempo specificato sia quello di durata di una intera partita;
  - hash: attiva/disattiva l'uso dell'hash file;
  - material: attiva/disattiva il controllo della patta per mancanza di materiale (quando nessuno dei due giocatori è in grado di dare scacco matto all'avversario);
  - random: se attivato rende più casuale la scelta della prossima mossa da parte del computer;
  - rcptr: attiva/disattiva il modo di ricattura, che consiste nella estensione della profondità di ricerca oltre l'orizzonte, fintanto che si continuano a catturare pezzi;

- controllo delle variabili:
  - debug: mostra il valore che un pezzo specificato assume nelle varie caselle della scacchiera;
  - Mwpawn, Mbpawn, Mwknight, Mbknight, Mwbishop, Mbbishop: visualizzano la tabella delle valutazioni statiche rispettivamente per pedone bianco e nero, cavallo bianco e nero, alfiere bianco e nero;
  - p: mostra il punteggio di ogni pezzo presente nella scacchiera. Il punteggio totale assegnato alla posizione è dato dalla somma di questi punteggi parziali;

  - test: esegue alcuni test di velocità per la generazione delle mosse e per la valutazione statica della posizione;

- input/output:
  - edit o set: permette all'utente di impostare una posizione inserendo un pezzo alla volta;
  - setup: permette di inserire una posizione specificando 8 linee di 8 caratteri, rappresentanti il contenuto delle varie caselle della scacchiera;
  - list: salva in un file predefinito le mosse della partita insieme ad alcune statistiche, come profondità, numero di nodi e tempo relativi ad ogni mossa;
  - save: salva una partita in un archivio permanente, col nome scelto dall'utente;
  - get: recupera una partita da un archivio permanente;
  - xget: legge un file di posizioni definito da xboard;

- gestione della partita:
  - quit o exit: esci da Gnuchess 4.0;
  - white: fa gestire al computer i pezzi bianchi;
  - black: fa gestire al computer i pezzi neri;
  - first: dice al computer di iniziare per primo;
  - force: permette all'utente di inserire le mosse per entrambi i colori;
  - go: dice al computer di eseguire la prossima mossa;
  - new: inizia una nuova partita;
  - remove: torna alla mossa precedente;
  - switch: scambia i colori tra l'utente e il computer;
  - undo: torna indietro di una semimossa;

8.2 Appendice B: partite giocate da Gnupar

Le seguenti partite sono state disputate dal giocatore artificiale parallelo con distribuzione bilanciata della conoscenza e con criterio di selezione "re_search", nella variante che assegna 1/3 del tempo alla selezione parallela e 2/3 alla ricerca successiva.
I dati statistici evidenziati riguardano il punteggio assegnato da Gnupar alla mossa eseguita (score), la profondità di ricerca raggiunta (depth), il numero di nodi visitati (nodes) e, per entrambi i giocatori, il tempo impiegato nell'esecuzione della mossa (time).

8.2.1 Partita da console

```
***************
Other vs Gnupar
***************
```

	score	depth	nodes	time		score	depth	nodes	time
e2e4	0	0	0	0	e7e5	0	Book	0	0
g1f3	0	0	0	5	b8c6	4	Book	0	0
d2d4	0	0	0	3	e5d4	-25	Book	0	0
f3d4	0	0	0	2	f8c5	3	Book	0	0
d4b3	0	0	0	1	c5b6	9	Book	0	0
a2a4	0	0	0	2	d8h4	22	4	36211	8
d1e2	0	0	0	19	g8f6	38	5	73723	22
b1c3	0	0	0	11	f6g4	-97	4	37495	8
g2g3	0	0	0	8	b6f2	267	4	61088	7
e1d1	0	0	0	17	f2g3	-271	4	40325	8
c1e3	0	0	0	38	g3e5	355	5	82173	21

```
Result: Black
```

8.2.2 Partita tramite ICS

```
***************
Gnupar vs Other
***************
```

	score	depth	nodes	time		score	depth	nodes	time
e2e4	10	Book	0	0	c7c5	0	0	0	6
g1f3	18	Book	0	0	d7d6	0	0	0	6
d2d4	4	Book	0	0	c5d4	0	0	0	4
f3d4	-61	Book	0	0	g7g6	0	0	0	5
f1b5	65	4	45972	20	c8d7	0	0	0	12
b1c3	36	6	56211	20	f8g7	0	0	0	6
o-o	52	5	59465	20	b8c6	0	0	0	12
c1e3	79	5	73934	20	g8f6	0	0	0	10

```
d4c6     77      5   55522    20  d7c6      0      0      0     7
b5c6     75      5   54397    21  b7c6      0      0      0     6
f2f4     71      5   50481    20   o-o      0      0      0    75
e4e5     48      5   73722    21  d6e5      0      0      0     5
d1d8    -17      6   70933    20  f8d8      0      0      0     4
e3c5    -23      6   93354    20  e5f4      0      0      0    23
c5e7    -17      4   54264     7  d8d2      0      0      0     3
f1f4    -72      4   51821     7  f6d5      0      0      0    18
c3d5     73      5   59757     7  c6d5      0      0      0     5
e7f6   -104      6   98266    20  d2c2      0      0      0    19
f6g7    -89      5   78738    20  g8g7      0      0      0     6
f4f2    -92      5   61581    20  a8c8      0      0      0     7
a1e1   -107      5   85903    20  d5d4      0      0      0     9
f2f1   -139      5   86340    20  c2b2      0      0      0    42
e1e7    143      5   53123     7  c8c2      0      0      0    17
f1f7     43      6   59649     8  g7h6      0      0      0     4
f7h7     43      6   60019     7  h6g5      0      0      0     4
h2h4   -140      4   54665     7  g5g4      0      0      0     7
e7e4   -139      5  119360    20  g4g3      0      0      0     5
e4e1  -9991      5   12251     9  c2g2      0      0      0    12
g1h1  -9993      3    2888     2  g2h2      0      0      0     7
h1g1  -9995      3    2027     1  b2g2      0      0      0     3
g1f1  -9997      2     101     0  h2h1  -9999      2     30     0

Result: Black
```

Bibliografia

[1] Abramson, B. Control strategies for two-player games. *ACM Computing Surveys 21*, 2 (Jun 1989), 137–161.

[2] Akl, S., Barnard, D., and Doran, R. Design, analysis and implementation of a parallel tree-search algorithm. *IEEE Transactions on pattern analysis and machine intelligence PAMI-4*, 2 (Mar 1982), 192–203.

[3] Althofer, I. Selective trees and majority systems: two experiments with commercial chess computers. *Advances in computer chess 6* (1991), 37–59.

[4] Bagley, J. D. The behaviour of adaptive systems which employ genetic and correlation algorithms. *Dissertation abstracts international 28*, 12 (1967).

[5] Bal, H. E., and Renesse, R. V. A summary of parallel alpha-beta search results. *ICCA Journal* (Sept 1986), 146–149.

[6] Baudet, G. M. The design and analysis of algorithms for asynchronous multiprocessors. Master's thesis, Carnegie-Mellon University, 1978.

[7] Beal, D. The 1992 QMW Uniform-Platform Autoplay Computer-Chess Tournament. *ICCA Journal 15*, 3 (September 1992), 162–167.

[8] Beal, D. Report on the QMW 1993 Uniform-Platform Computer-Chess Championship. *ICCA Journal 16*, 3 (September 1993), 166–170.

[9] Bratko, I., and Kopec, D. The bratko-kopec experiment: a comparison of human and computer performance in chess. *Advances in computer chess 3* (1982), 57–72.

[10] Carriero, N., and Gelernter, D. *How to write parallel programs. A first course.* The MIT Press, 1990.

[11] Ciancarini, P. *I giocatori artificiali di scacchi.* Mursia, 1992.

[12] Ebeling, C. All the right moves: a VLSI architecture for chess. *The MIT Press* (1986).

[13] EDWARDS, D., AND HART, T. The alpha-beta heuristic. Tech. rep., Computer Science Dept., Massachussets Institute of Technology, Cambridge, 1963.

[14] ELO, A. E. *The rating of chessplayers: past and present*, 2 ed. AARCO Publid shing, INC., 1986.

[15] FERGUSON, C., KORF, R., AND POWLEY, C. Depth-first heurustic search on a SIMD machine. *Artificial Intelligence 60* (1993), 199–242.

[16] FISHBURN, J., AND FINKEL, R. Parallelism in alpha-beta search. *Artificial Intelligence 19* (1982), 89–106.

[17] GILLOGLY, J. J. Performance analysis of the technology chess program. Research sponsored CMU-CS-78-109, Carnegie-Mellon University, Mar 1978.

[18] GOLDBERG, D. E. Genetic algorithms in search, optimization and machine learning. *Addison-Wesley* (1989).

[19] HOLLAND, J. H. Adaptation in natural and artificial systems. *The University of Michigan Press* (1975).

[20] KNUTH, D., AND MOORE, R. An analysis of alpha-beta pruning. *Artificial Intelligence 6* (1975), 293–326.

[21] LANG, K., AND SMITH, W. A test suite for chess programs. *ICCA Journal 16*, 3 (September 1993), 152–161.

[22] MARSLAND, M. A., AND CAMPBELL, M. Parallel search of strongly ordered game trees. *ACM Computing Surveys 14*, 4 (Dec 1982), 533–551.

[23] MARSLAND, T. A., OLAFSSON, M., AND SCHAEFFER, J. Multiprocessor tree-search experiments. *Advances in computer chess 4* (1985), 37–51.

[24] MARSLAND, T. A., AND POPOWICH, F. Parallel game-tree search. *IEEE Transactions on pattern analysis and machine intelligence PAMI-7*, 4 (Jul 1985), 442–452.

[25] SCHAEFFER, J. Experiments in search and knowledge. Technical Report TR 86-12, University of Alberta, Edmonton, Alberta, Canada, July 1986.

[26] SCHAEFFER, J. Speculative computing. *ICCA Journal* (Sept 1987).

[27] SCIENTIFIC COMPUTING ASSOCIATES INC. *C-Linda reference manual.* New Haven, Connecticut, 1990.

[28] SHANNON, C. E. Programming a computer for playing chess. *Philosophical magazine 41* (1950), 256–275.

[29] TOZZI, M. Progetto e realizzazione di un programma distribuito di visita di alberi di gioco. Master's thesis, Università degli studi di Pisa, 1993.

[30] TUNSTALL-PEDOE, W. Genetic algorithms optimizing evaluation functions. *ICCA Journal 14*, 3 (1991), 119–128.

[31] VON NEUMANN, J., AND MORGENSTERN, O. Theory of games and economic behaviour. Tech. rep., Princeton University, Princeton, NJ, 1944.