Indice generale

Capitolo 0
Introduzione

0.1 Intelligenza artificiale = conoscenza + ricerca

Il termine intelligenza artificiale sta ad indicare quella branca della scienza che si occupa dello studio dei problemi decisionali e della loro soluzione attraverso il calcolatore. Quest'ultimo è dunque astratto ad una entità capace di operare autonomamente delle scelte.

Conoscenza e ricerca sono le componenti fondamentali di un sistema capace di prendere decisioni: ciascuna di esse è responsabile, in modo diverso, delle sue prestazioni. In particolare la conoscenza determina la qualità delle scelte, mentre la ricerca stabilisce quanto velocemente esse sono ottenute.

Il presente lavoro intende limitare il suo dominio di indagine ad una sotto-classe molto significativa di sistemi "intelligenti": quella dei giocatori artificiali.

Il tipo di giochi che sarà preso in considerazione è quello che vede coinvolti 2 avversari e gode delle proprietà di informazione perfetta (in ogni fase del gioco sono note ai giocatori tutte le mosse legali) e somma-zero (ciò che un giocatore perde è equivalente al guadagno dell'altro).

In questo ambito per giocatore artificiale deve intendersi un sistema capace di ricoprire autonomamente il ruolo di uno dei giocatori e di operare delle scelte (nel rispetto delle regole del gioco) mirate ad un vantaggio locale (miglioramento dal suo punto di vista di certe condizioni del gioco) o assoluto (vittoria di una partita) nei confronti dell'avversario.

Per questa classe di sistemi la ricerca è intesa come lo sviluppo delle linee di gioco future con lo scopo di valutare e confrontare quali conseguenze possono determinare scelte (mosse) diverse operate al livello dello stato corrente del gioco. La struttura che modella questo tipo di ricerca è un albero (detto di gioco) i cui nodi rappresentano posizioni future di gioco raggiunte attraverso mosse legali (gli archi) [Abr89]. La profondità di questo albero quantifica la proiezione nel futuro del giocatore.

La conoscenza del dominio è presente, in un giocatore artificiale, in duplice veste [Ber82]:

• per guidare la ricerca suggerendo in quale ordine approfondire mosse alternative (conoscenza dirigente)

• per stabilire quanto è "appetibile" (o meno) il raggiungimento di una certa posizione (conoscenza terminale).

Shannon ha proposto una classificazione anche per la ricerca [Sha50]:

• con forza-bruta: sono esaminate fino ad una certa profondità tutte le possibili scelte

• selettiva (o guidata dalla conoscenza): sono esplorate solo quelle alternative che appaiono fornire (con elevata probabilità) il maggiore contributo per la determinazione della soluzione.

La finalità principale di questo lavoro è lo studio in ambiente distribuito di vari aspetti di conoscenza e ricerca e delle loro interazioni reciproche. Utilizzando il dominio applicativo del gioco degli scacchi sarà sperimentata la distribuzione di alcuni metodi di ricerca con forza bruta e proposto (e sperimentato) un nuovo approccio allo sviluppo di giocatori paralleli basato sul concetto di "distribuzione della conoscenza".

Gli scacchi costituiscono un terreno ideale per lo studio di tali problematiche poiché per questo gioco le prestazioni di un giocatore artificiale sono fortemente dipendenti sia dal suo algoritmo di ricerca che dalla conoscenza del domino di cui esso dispone; in seguito saranno presentati altri argomenti a sostegno di questa scelta.

Le prestazioni di un giocatore artificiale sono misurate dalla sua forza, dalla qualità del suo gioco: per quali motivi scaturisce l’esigenza di giocatori paralleli? come può contribuire la distribuzione della sua ricerca e della sua conoscenza a migliorare le prestazioni di un giocatore?

Le righe che seguono intendono dare una risposta intuitiva a tali quesiti attraverso un breve e suggestivo viaggio nell’esperienza acquisita negli ultimi decenni riguardo l’importanza relativa di conoscenza e ricerca in un giocatore artificiale.

Nel mondo dei giochi con 2 avversari e a somma-zero è riconosciuto che i migliori programmi utilizzano tecniche di ricerca con forza bruta: scacchi, dama [Sam60] e othello [Ros81] ne sono esempi. La ricerca con forza bruta ha il vantaggio di ridurre la conoscenza del dominio richiesta.

La valutazione di una posizione del gioco, sia che sia operata da un uomo che da un calcolatore, è il tentativo di indicare quanto essa sia favorevole in considerazione delle sue prospettive future. La ricerca a maggiori profondità può sostituire queste predizioni attraverso un esplicito attraversamento del futuro mediante il quale sono risolte alcune delle incertezze.

Estendendo all’estremo questa considerazione, un giocatore capace di operare ricerche ad illimitate profondità ha bisogno della sola conoscenza necessaria ad individuare posizioni di vittoria, sconfitta e parità. Si viene così a delineare l’apparente paradosso che può essere creato un programma capace di giocare perfettamente disponendo di conoscenza pressoché nulla del dominio. Purtroppo le dimensioni degli alberi esplorati dagli algoritmi di ricerca con forza bruta crescono esponenzialmente con la profondità ed in generale non è quindi possibile raggiungere gli stati terminali del gioco.

La profondità di ricerca costituisce l’orizzonte della visione futura del giocatore: quale influenza può avere la sua forzata limitazione?

Newborn ha stimato che raddoppiando la potenza di calcolo di un giocatore artificiale (attraverso calcolatori più veloci o migliori algoritmi di ricerca) aumenta di una quantità costante la sua forza [New79]. Chiaramente un programma più veloce può esaminare un maggior numero di nodi e quindi raggiungere profondità maggiori dell'albero.

Da queste considerazioni emerge l'importanza dello sviluppo di algoritmi di ricerca paralleli in quanto capaci di garantire una maggiore velocità di calcolo e quindi di estendere l'orizzonte dell'analisi delle continuazioni del gioco.

Questo approccio non è di per se soddisfacente:

• la legge di corrispondenza fra la velocità di calcolo e le prestazioni del giocatore diviene meno evidente quando quest'ultime raggiungono quelle dei migliori giocatori umani [Tho82].

• il guadagno (speedup) che gli algoritmi paralleli di ricerca su alberi di gioco garantiscono rispetto a quelli sequenziali tende a stabilizzarsi con l'aumentare del numero di processori: Finkel e Fishburn hanno dimostrato che in condizioni ottime esso è $o(\sqrt{N})$ dove N è il numero di processori [FinFis82].

Un approccio duale a quello presentato è quello di fare affidamento su una notevole conoscenza del dominio che permetta di valutare accuratamente le posizioni del gioco utilizzando una ricerca minima; questo è anche lo schema che guida le scelte dei giocatori umani. Portando questa idea al suo estremo un giocatore con una profonda conoscenza del gioco potrebbe non avere bisogno di alcuna ricerca.

I giochi più complessi, tuttavia, richiedono un'enorme quantità di conoscenza. Si consideri ad esempio il gioco degli scacchi:

• PARADISE è un programma che usa ben 200 regole di produzione solo per pianificare alcune conquiste di materiale o situazioni di matto [Wil79];

• la soluzione corretta di alcuni problemi di finale necessita di decine di regole [Sch86]:

• re e pedone contro re (KP vs. K) richiede 38 regole

• re, pedone e torre in settima riga contro re e torre (KP/r7 vs. KR) richiede 110 regole

• re e torre contro re e cavallo (KR vs. KN) è risolto con 65 regole.

Indicare quantità di regole senza specificarne il significato è certamente non significativo; tuttavia queste statistiche assolvono il compito di illustrare l'enorme complessità già di un piccolo sottoinsieme del gioco degli scacchi.

Poiché un giocatore artificiale non può dunque disporre di tutta la conoscenza necessaria ad ottenere un gioco perfetto resta aperto il problema di determinare quale conoscenza ha maggiore importanza.

Schaeffer ha dimostrato che l'importanza relativa di differenti categorie di conoscenza è fortemente influenzata dallo stato del gioco [Sch86]: nel gioco degli scacchi, ad esempio, nella fase di

mediogioco è fondamentale il controllo delle case centrali, ma l'importanza di questa conoscenza scema nel finale di partita.

In questo ambito il significato di distribuire la conoscenza è di disporre di più giocatori con diversa conoscenza del dominio che cooperano suggerendo ciascuno la migliore mossa rispetto al proprio punto di vista; emergono evidenti due vantaggi di questo approccio:

• l'applicazione di conoscenza ha un suo costo computazionale e quindi deve essere limitata nella sua quantità; disponendo però di più giocatori è possibile estendere la quantità di conoscenza complessiva suddividendola fra di essi.

• non esiste un ordinamento assoluto per importanza relativa delle categorie di conoscenza. Quello che è fissato all'interno di un generico giocatore artificiale è normalmente il risultato di esperienza e di tecniche "trial-and-error", cioè di approssimazioni per tentativi della soluzione migliore: nulla garantisce, però, che il risultato di questo procedimento sia effettivamente quello ottimale. La distribuzione della conoscenza permette di avere anche giocatori che pur disponendo di conoscenza comune, attribuiscono alle componenti di essa importanza relativa differente.

Deve essere sottolineata l'assoluta novità di questo approccio nel progetto di un giocatore artificiale parallelo (in [Alt91] è tuttavia descritta una "rudimentale" anticipazione di questa idea). La sua validità deve dunque essere confermata dalla reale sperimentazione: questo costituisce uno degli obbiettivi principali di questo lavoro.

Allo stato dell'arte nessuna delle due soluzioni estreme descritte (giocatore artificiale con sola ricerca o con sola conoscenza) può condurre ad un sistema capace di concorrere con i migliori giocatori umani: l'unico percorso praticabile è quindi quello di un approccio intermedio in cui entrambe le componenti coesistono. In analogia a questa considerazione è lecito chiedersi:

ha significato combinare in un unico giocatore parallelo sia la distribuzione della ricerca che della conoscenza?

è questo il futuro della ricerca in questo ramo dell'intelligenza artificiale?

Il presente lavoro intende, attraverso un'analisi separata delle due forme di parallelismo, gettare le basi per il progetto di un unico giocatore parallelo che scaturisca da questa duplice distribuzione.

0.2 L'importanza degli scacchi come dominio di applicazione

Il gioco degli scacchi costituisce un eccellente dominio per il confronto e lo sviluppo di studi e metodi aventi per oggetto sistemi "intelligenti".

In [Sch86] e [HLMSW92] sono discussi i motivi di tale convinzione:

• la maggior parte dei sistemi esperti è caratterizzata da un piccolo e ben definito insieme di condizioni di ingresso e di uscita. Per

questa classe di sistemi possono essere enumerate tutte le possibili soluzioni: la loro "intelligenza" può quindi essere implementata attraverso una semplice consultazione di tabelle predefinite. La complessità inerente di un sistema esperto capace di giocare a scacchi emerge dal numero di possibili stati del gioco: $\approx 10^{120}$ [DGr65]. Essi sono decisamente troppi per raccogliere esperienza che permetta un gioco perfetto per ognuno di essi. Per superare questo problema il giocatore artificiale deve usufruire di conoscenza generale che gli permetta di descrivere e "risolvere" il maggior numero possibile di stati. Il risultato è una conoscenza relativamente inesatta (in questo caso si parla di euristica) la quale è fonte di errore. Questa proprietà è caratteristica di tutti i sistemi esperti più complessi e ciò conferisce generalità al dominio degli scacchi.

• il gioco è da sempre considerato di "intelligenza"; esso costituisce una sfida intellettuale che gli ultimi 200 anni di intensa ricerca non hanno sminuito di interesse ed esaurito l'analisi di tutti i suoi aspetti;

• le regole del gioco sono ben definite: ciò di fa esso un utile dominio per il test di metodi applicati normalmente a problemi non discreti (probabilistici);

• il dominio può essere facilmente decomposto in sottodomini per isolare alcuni problemi (ad es. si può pensare ad un sistema capace di giocare bene i finali di partita);

• esiste il sistema ELO [CFC84], una metrica internazionale di valutazione delle prestazioni che permette di attribuire un valore assoluto alle qualità di un certo giocatore;

• molta della conoscenza scacchistica deve essere ancora valutata negli effetti che produce quando incorporata in un giocatore artificiale;

• il gioco degli scacchi è molto noto e ciò permette al ricercatore di valutare direttamente i risultati del sistema da lui sviluppato; ciò non è sempre possibile: in molte applicazioni lo studioso non ha familiarità con il dominio di applicazione e deve fare spesso affidamento sull'opinione di esperti.

0.3 Gli strumenti

0.3.1 La base teorica

Lo strumento fondamentale cui farà affidamento questo lavoro è l'esperienza acquisita negli ultimi decenni dal mondo scientifico per ciò che riguarda i sistemi "intelligenti" ed in particolare i giocatori artificiali.

La maggior parte degli studiosi ha concentrato l'attenzione sulla componente di ricerca dei sistemi. In questo ambito i metodi di ricerca con forza bruta sono stati i più esplorati ed il risultato più importante di tanto impegno è l'algoritmo $\alpha\beta$ [KnuMoo75]. La sua caratteristica è di eseguire una visita dell'albero di gioco in ordine depth-first e di operare dei tagli

dello spazio di ricerca (in modo analogo agli algoritmi branch-and-bound). Per taglio si intende la capacità di individuare, prima della loro esplorazione, sottoalberi che certamente non contengono la soluzione. Il vantaggio di eseguire tagli riduce la complessità della ricerca, anche se essa rimane esponenziale rispetto alla profondità massima da essa raggiunta.

Le proprietà degli algoritmi di ricerca selettiva non sono state ancora sufficientemente investigate poiché di norma essi sono dipendenti dal dominio e i modelli matematici che li descrivono generalmente complessi e di tipo probabilistico. L'algoritmo dei numeri di cospirazione di McAllester [McA85;Sch90] costituisce tuttavia un esempio convincente per efficacia di algoritmo di ricerca selettiva indipendente dal dominio.

Come già precisato, l'attenzione di questo lavoro si soffermerà sullo studio di giocatori artificiali paralleli. In questo settore i ricercatori hanno impegnato i maggiori sforzi nello studio della parallelizzazione dell'algoritmo $\alpha\beta$. Tale problema si è rivelato estremamente difficile: mentre alcuni risultati di simulazioni hanno promesso buone prestazioni [Lin83; AkBaDo82], alcune implementazioni reali hanno rivelato modesti speedup [MaOlSc85; New85]. Al di là di un certo numero di processori le prestazioni si stabilizzano fino al punto in cui l'introduzione di nuove risorse di elaborazione produce effetti negativi. I principali problemi che affliggono la parallelizzazione $\alpha\beta$ sono l'overhead di sincronizzazione, cioè il costo che deriva dai tempi in cui i processori sono inoperosi e l'overhead di ricerca, conseguenza della deficienza di informazioni nell'ambiente globale ai processori.

Fra gli algoritmi $\alpha\beta$ paralleli quello che è emerso come più vantaggioso è PVSplit [MarCam82]: è relativamente semplice da capire ed implementare e ha garantito risultati soddisfacenti in presenza di un piccolo numero di processori [MaOlSc85; New85].

Allo stato dell'arte rimane tuttavia aperta la discussione su quale sia la migliore implementazione di $\alpha\beta$ in presenza di decine o centinaia di processori [Sch89].

Gli studi nel campo della conoscenza sono inerentemente dipendenti dal dominio di applicazione. In questo ambito i problemi più importanti sono la scelta della conoscenza di cui deve disporre un sistema e la determinazione dell'importanza relativa delle "porzioni" che la compongono. Tali problematiche non rientrano nel campo di interesse del presente lavoro il quale intende suggerire metodi di "gestione" (in particolare di distribuzione) della conoscenza indipendenti dal dominio. Dovendo però sperimentare

queste idee e dare significato concreto a tali esperimenti, si dovrà comunque "manipolare" conoscenza appartenente ad un dominio reale (gli scacchi). L'approccio sarà quello di considerare come validi i risultati di alcuni ricercatori [Sch86] senza questionare sulle scelte e i metodi che li hanno originati.

L'unico riferimento per lo sviluppo dell'idea di distribuzione della conoscenza è il lavoro di Althöfer [Alt91]: egli ha eseguito alcuni esperimenti con giocatori paralleli di scacchi costituiti da giocatori sequenziali commerciali che cooperano suggerendo ciascuno la propria migliore mossa e selezionando quale scelta finale la mossa maggiormente proposta. In esso la distribuzione della conoscenza è implicita nella diversità dei singoli giocatori. Tale distribuzione è però casuale, non scaturisce cioè dall'applicazione di criteri e metodi generali: il concepimento, l'analisi e la sperimentazione di questi sarà invece uno dei temi di questo lavoro.

0.3.2 Il supporto a implementazione e sperimentazione

Il supporto per l'implementazione e la sperimentazione delle idee che saranno proposte è costituito dai seguenti strumenti:

- l'architettura parallela
- il linguaggio di programmazione
- il giocatore sequenziale di riferimento

L'architettura parallela è una rete locale di 11 SUN 4 Sparcstation collegate in Ethernet.

Le caratteristiche principali di questa architettura che dovranno essere tenute in considerazione in sede di progetto dei programmi e di discussione dei risultati sono:

- l'assenza di memoria condivisa e quindi la possibilità di cooperare fra processori unicamente attraverso lo scambio fisico di messaggi attraverso la rete;

- l'accoppiamento medio fra i processori: l'affidabilità e la velocità con cui avvengono le comunicazioni inter-processore si pongono a metà fra quelle elevate di architetture a parallelismo massiccio e quelle decisamente inferiori di reti esterne. Il costo di una comunicazione è dell'ordine del millisecondo.

Il linguaggio di programmazione distribuita utilizzato è C-Linda: si tratta di una estensione del linguaggio sequenziale C con le primitive del modello Linda di cooperazione fra processi [CarGel90].

Linda è un nuovo modello di programmazione parallela i cui meccanismi di comunicazione sono basati sul concetto di strutture dati distribuite, cioè che possono essere manipolate

simultaneamente da più processi. In Linda esse sono implementate attraverso un modello di memoria denominato spazio delle tuple [CarGel89]. Questo è una memoria globale condivisa fra i processi (anche se la sua implementazione non richiede necessariamente la condivisione di memoria fisica).

Gli elementi dello spazio delle tuple sono una sequenza ordinata di valori chiamati tuple. Su di essi sono definite le seguenti operazioni:

- out: deposito di una nuova tupla
- in: lettura e rimozione di una tupla
- rd: lettura (senza rimozione) di una tupla

La comunicazione fra i processi è dunque mediata dallo spazio delle tuple. La sincronizzazione fra processi è attuata dall'attesa della disponibilità delle tuple desiderate: le primitive in e rd, infatti, sono bloccanti, cioè sospendono il processo che le invoca fino a che nello spazio delle tuple non compare (per effetto di un altro processo) la tupla attesa.

Linda fornisce una primitiva molto semplice, chiamata eval, per la creazione di un nuovo processo sequenziale; la particolarità dei processi così creati è di costituire tuple attive: al momento della loro terminazione essi generano un risultato che è parte della stessa tupla che è divenuta ora passiva e può essere letta e rimossa come qualsiasi altra.

Linda non costituirà semplicemente uno strumento di lavoro, ma sarà anche oggetto di analisi. Uno dei principali obbiettivi di questa tesi, infatti, è valutare il suo modello di comunicazione quando applicato in un dominio reale. Particolare attenzione sarà rivolta alle proprietà espressive e di efficienza di questo linguaggio.

Le prestazioni dei giocatori paralleli che scaturiranno sia dalla distribuzione della ricerca che della conoscenza saranno confrontate con quelle di un giocatore artificiale sequenziale che quindi costituirà il termine di misura e confronto per tutti i miglioramenti incorporati in essi. La scelta di tale giocatore è caduta su GnuChess, un programma di scacchi prodotto dalla OSF (progetto GNU). I motivi di questa scelta sono molteplici:

- GnuChess è relativamente semplice: il suo codice si compone di circa 4000 linee in C. Un sufficiente livello di confidenza con esso può quindi essere ottenuto in tempi piuttosto brevi (approssimativamente 15 giorni).

- le prestazioni del giocatore sono notevoli: nonostante la sua semplicità è caratterizzato da una considerevole qualità di gioco. Quanto detto trova conferma nelle vittorie da esso riportate nelle edizioni '92 e '93 del torneo annuale su piattaforma stabile[1] fra giocatori artificiali di scacchi.

• il programma è di pubblico dominio: ciò fa di esso un terreno comune di confronto e di scambio fra molti dei ricercatori del settore.

• la struttura del programma è abbastanza modulare: questa proprietà permette il suo utilizzo come libreria di funzioni. In questa veste GnuChess (in particolare alcune delle sue funzionalità) costituirà l'interfaccia fra gli algoritmi sviluppati in questo lavoro, in generale indipendenti dal dominio di applicazione, ed il dominio in cui essi saranno sperimentati: il gioco degli scacchi.

0.4 La struttura della tesi

La prima parte di questa dissertazione (Capitoli 1-4) sarà dedicata ad una descrizione più attenta degli strumenti necessari allo sviluppo ed alla sperimentazione di nuovi metodi di distribuzione della ricerca e della conoscenza all'interno di un giocatore artificiale; la discussione di quest'ultimi e la presentazione dei relativi risultati sperimentali sarà invece oggetto della seconda parte della tesi (Capitoli 5-6).

In particolare nel Capitolo 1 si intende prendere conoscenza dello stato dell'arte nel campo della ricerca sequenziale degli alberi di gioco. Sarà inclusa la presentazione del concetto di albero di gioco come modello matematico e di alcuni algoritmi di visita dell'albero. Particolare enfasi sarà posta nella introduzione degli algoritmi con forza bruta: minimax, $\alpha\beta$ e le sue varianti. Saranno inoltre passate in rassegna alcune delle euristiche di ricerca mirate a migliorare l'efficienza di tali algoritmi. Quale rappresentante dell'approccio selettivo sarà descritto l'algoritmo dei numeri di cospirazione di McAllester.

Il capitolo 2 ha per argomento la descrizione strutturale di GnuChess. Essa si articolerà attraverso l'analisi della rappresentazione discreta degli stati del gioco, la presentazione dell'algoritmo e delle euristiche di ricerca ed infine delle modalità di gioco e dell'interfaccia di i/o. Particolare attenzione sarà dedicata alla funzione di valutazione statica, cioè la componente che raccoglie la conoscenza terminale del giocatore: sarà infatti proposta una sua decomposizione logica che costituirà nel Capitolo 6 la base per la sperimentazione dei metodi di distribuzione della conoscenza. Lo studio di un giocatore reale (quale GnuChess è) rappresenterà inoltre un'utile verifica di alcune delle indicazioni teoriche raccolte nel Capitolo 1.

Il Capitolo 3 sarà dedicato alla presentazione di Linda, il linguaggio di programmazione distribuita attraverso il quale saranno descritti gli algoritmi classici di ricerca (Capitolo 4) e che costituirà strumento di progetto ed implementazione delle nuove idee prima citate (Capitoli 5 e 6). Sarà presentato lo spazio delle tuple, cioè il modello di comunicazione su cui tale linguaggio è basato, e gli operatori su esso definiti. La discussione dell'operatore eval di creazione dei processi suggerirà l'introduzione del modello

master-worker, vale a dire il paradigma di programmazione concorrente che sarà preso a riferimento nei Capitoli 5 e 6 nel progetto della cooperazione fra processi. Verrà inoltre presentato Network C-Linda, cioè l'istanza di Linda impiegata in questo lavoro, e fornite indicazioni sullo stile di programmazione in Linda attraverso gli esempi:

- di alcune strutture dati distribuite
- della descrizione in Linda di varie forme di comunicazione
- di sincronizzazione fra processi.

Ad epilogo del capitolo verranno accennate le problematiche di implementazione del linguaggio con particolare attenzione alla realizzazione su rete locale (l'architettura parallela di riferimento per questo lavoro).

Il Capitolo 4 descrive lo stato dell'arte nell'ambito della ricerca parallela degli alberi di gioco. Saranno presentate diverse classi di soluzioni:

- parallelismo nell'esecuzione di attività che riguardano i nodi individualmente
- ricerca con finestre parallele (parallel aspiration search)
- mapping dello spazio di ricerca nei processori
- decomposizione dell'albero di gioco

Molto spazio sarà dedicato agli algoritmi di decomposizione i quali costituiranno oggetto di approfondimento e sperimentazione nel Capitolo 5. In particolare sarà formalizzata una nuova classificazione di tali metodi attraverso la quale si intende stabilire maggiore "ordine" all'interno di questo insieme estremamente complesso di algoritmi e fissare una nuova terminologia che favorisca la distinzione fra le loro proprietà salienti. Successivamente saranno illustrati i motivi di degrado di questi algoritmi e alcuni esempi fra i più significativi: l'algoritmo base, mandatory work first, PVSplit e le sue varianti. Sarà infine discussa l'efficacia in ambiente distribuito delle euristiche di ricerca utilizzate con successo in ambiente sequenziale.

Il Capitolo 5 concerne il progetto e la sperimentazione di alcuni metodi di distribuzione della ricerca. In particolare sarà presa in considerazione la realizzazione in Linda di alcuni algoritmi di decomposizione dell'albero di gioco. A partire dall'algoritmo base saranno sviluppati algoritmi via via più complessi in termini di granularità[2]. Oltre a conseguire una maggiore efficienza questi miglioramenti saranno mirati a capire quali classi di problemi possono essere risolti in Linda senza pregiudicare le prestazioni. Sarà inoltre sviluppato un algoritmo "generale" di decomposizione il cui principale scopo non è quello dell'efficienza, ma di costituire una struttura estremamente flessibile adatta per lo studio "rapido" di qualsiasi metodo di decomposizione: la proprietà dell'algoritmo ideato, infatti, è di essere "parametrico" rispetto al criterio di decomposizione, cioè di avere isolato tale criterio in un unico

modulo rendendo così trasparente la gestione distribuita della ricerca ed in particolare la cooperazione fra i processi.

Il Capitolo 6 introduce il concetto nuovo di "distribuzione della conoscenza". Sarà discussa l'importanza della conoscenza nel dominio dei giochi e presentati alcuni risultati riguardanti l'attribuzione di importanza relativa a diverse categorie di conoscenza scacchistica [Sch86]. Di seguito sarà spiegata la struttura di un giocatore parallelo costituito da più istanze dello stesso giocatore sequenziale (GnuChess), ma con diversa conoscenza terminale. L'attribuzione di conoscenza ai diversi giocatori avviene in rispetto di un criterio chiamato appunto di distribuzione. I giocatori cooperano suggerendo ciascuno la propria mossa e determinando la migliore, fra quelle suggerite, sulla base di un criterio di selezione. Saranno ideati e sperimentati diversi criteri di distribuzione e di selezione. In particolare verranno suggerite metriche per la valutazione separata di tali criteri permettendo così di scinderne gli effetti sulle prestazioni complessive del giocatore parallelo. In conclusione sarà discussa la possibilità di distribuzione della conoscenza dirigente intendendo con questo termine la cooperazione di giocatori sequenziali che adottano differenti algoritmi ed euristiche di ricerca.

Il capitolo 7 presenterà un sommario del lavoro svolto e offrirà suggerimenti per le ricerche future.

Capitolo 1
Algoritmi di ricerca sequenziale di alberi di gioco

1.1 Introduzione

La teoria dei giochi è una branca relativamente giovane della matematica. La sua finalità è lo studio di problemi decisionali che coinvolgono in generale più persone (o agenti, o partiti).

Nella classe di problemi menzionata le persone coinvolte (solitamente chiamate giocatori) hanno interessi differenti e mirano ad ottenere soluzioni diverse: esse competono l'una contro l'altra per ottimizzare il proprio profitto.

Esempi di problemi decisionali con più partiti non sono soltanto giochi da tavolo quali gli scacchi o il bridge, ma anche problemi contrattuali fra un venditore ed un acquirente oppure di competizione sul mercato da parte di più aziende. La teoria dei giochi, dunque, non limita il suo dominio applicativo al solo ambito dei giochi, ma fornisce un insieme di strumenti (nella forma di modelli e risultati matematici) che trovano applicazione in molte altre discipline come le scienze economiche, sociologiche, sociobiologiche e politiche [Thu92].

L'attenzione degli studiosi di questa disciplina si è concentrata, fin dai suoi albori, sull'analisi di una classe particolare di problemi che prende il nome di giochi di due persone, a somma zero e con informazione perfetta; i componenti di tale classe sono accomunati da medesime caratteristiche [vNeMor44]:

- il numero dei giocatori è 2
- ciò che un giocatore perde equivale al guadagno dell'altro (somma zero)
- tutte le mosse legali sono note ad entrambi i giocatori in ogni fase della partita (informazione perfetta).

Gli esempi più rappresentativi di questo insieme sono scacchi, dama, othello, tic-tac-toe, ecc.

Il dominio di indagine del presente lavoro è proprio questa classe di giochi.

In questo capitolo è introdotto il modello matematico discreto su cui è basata la teoria di questi problemi: l'albero di gioco [Sha50]. È inoltre presentata una rassegna di algoritmi e risultati teorici sviluppati sia in ambiente sequenziale che parallelo.

1.2 Alberi di gioco

Si definisce albero di gioco una struttura ad albero i cui nodi rappresentano stati del gioco e ciascun arco una trasformazione di tale stato originata dell'esecuzione di un'azione (o mossa) legale, cioè ammessa dalle regole del gioco.

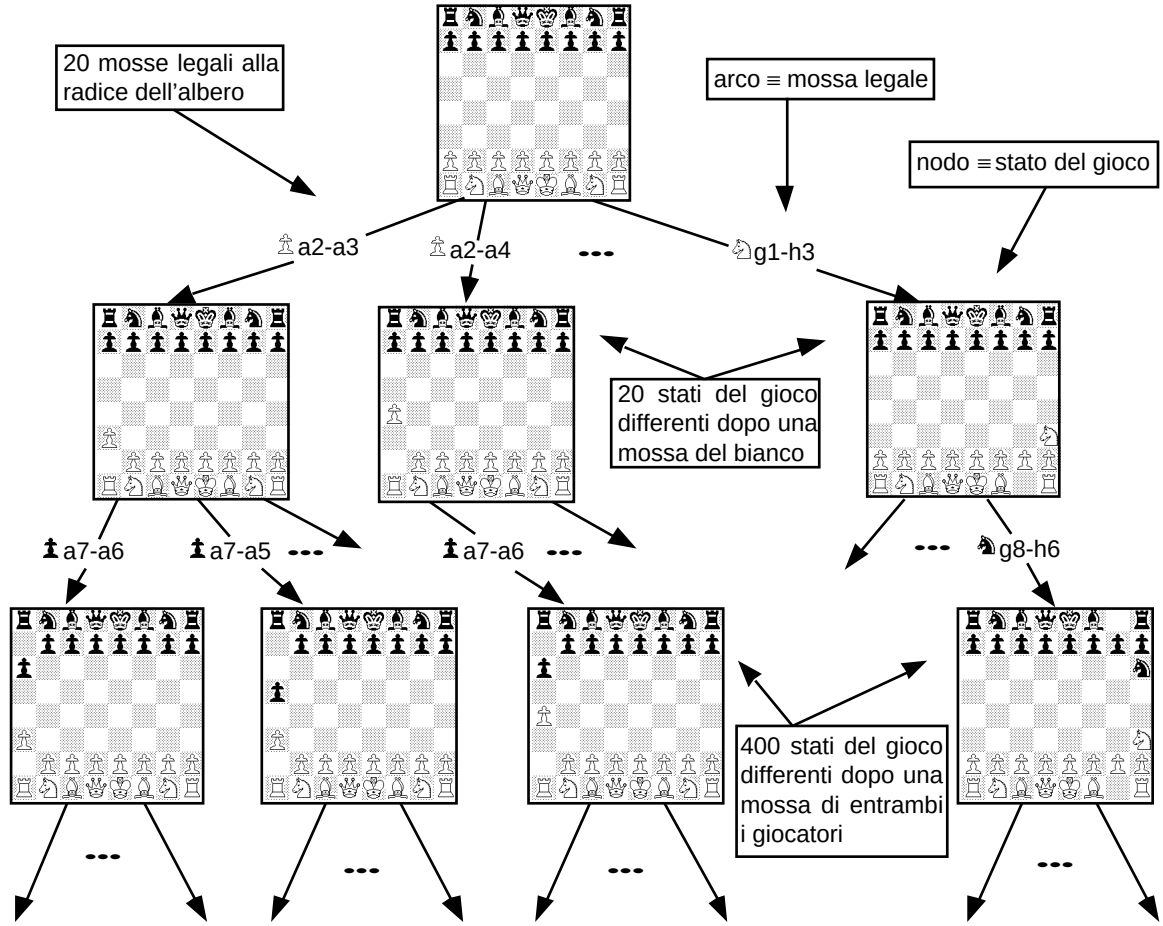


Fig. 1.1 Albero di gioco negli scacchi

In Fig. 1.1 è illustrato l'albero di gioco degli scacchi:

• il nodo radice definisce l'impostazione iniziale della scacchiera

• gli archi che lo collegano ai nodi successori individuano l'insieme di aperture legali

• ogni cammino a partire dalla radice rappresenta una differente partita.

Si supponga che il gioco abbia realmente raggiunto un certo stato (corrente) e si consideri il sottoalbero dell'albero di gioco la cui radice è il nodo che rappresenta questo stato. Tale sottoalbero descrive tutti i possibili sviluppi futuri della partita: anche questo è un albero di gioco. Normalmente quando si parla "dell'albero di gioco" ci si riferisce implicitamente proprio a quello relativo allo stato corrente poiché descrive completamente ciò che è importante per un giocatore: il futuro del gioco.

Più formalmente, l'albero di gioco è dunque una struttura definita in maniera ricorsiva consistente di un nodo radice rappresentante lo stato iniziale e di un insieme finito di archi indicanti mosse legali in quel nodo. Gli archi puntano a potenziali stati successivi ognuno dei quali, a sua volta, è la radice di un nuovo albero di gioco.

Un nodo che non ha archi uscenti è un nodo terminale o foglia ed individua una posizione per la quale non vi sono mosse legali. Se lo

stato corrente è una foglia, allora il gioco ha termine. Ad ogni foglia è associato un valore (ad esempio: VITTORIA, SCONFITTA o PAREGGIO) che sintetizza l'esito finale della partita.

Dato un nodo, il numero di archi uscenti ne definisce il fan-out, mentre la distanza dalla radice ne individua la profondità. Il termine fattore di diramazione è usato per indicare il valore medio del fan-out per un dato albero di gioco.

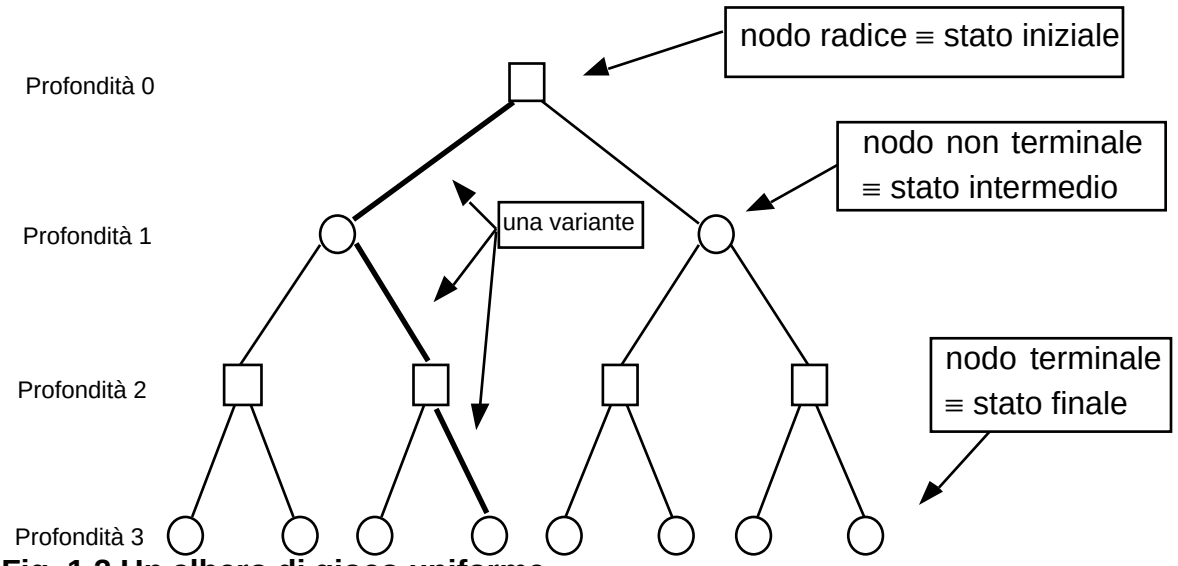


Fig. 1.2 Un albero di gioco uniforme

Un albero di gioco si dice uniforme se tutti i nodi non terminali hanno lo stesso fan-out e tutti i nodi foglia sono alla stessa profondità. Un albero uniforme con fattore di diramazione w e profondità d contiene un totale di $\sum_{i=1}^{d} w^d = \frac{w^{d+1}-1}{w-1}$ nodi, w^d dei quali sono terminali.

L'albero di gioco uniforme di Fig. 1.2 ha un fattore di diramazione 2 e profondità 3.

Un cammino dal nodo radice ad uno dei nodi terminali identifica una particolare istanza di partita ed è chiamato variante.

Il modello albero di gioco è discreto e quindi molto adatto ad essere rappresentato in un calcolatore. Ciò che si vuole ottenere è di rendere un computer capace di giocare in maniera autonoma, possibilmente ai livelli dei giocatori umani più esperti; vedremo nel seguito come questa possibilità dipenda da molti fattori, principalmente dalla potenza di calcolo a disposizione.

Come può un calcolatore operare una decisione di natura strategica quale, in un certo stato del gioco, l'esecuzione della mossa più vantaggiosa ? Esso è istruito (programmato) a selezionare la mossa "migliore" secondo una strategia di controllo.

1.2.1 Algoritmo minimax: una strategia di controllo

Per strategia di controllo si intende una procedura di controllo del flusso del gioco che suggerisce ad un giocatore quale mossa scegliere.

L'algoritmo minimax è l'esempio più importante di strategia di controllo per la classe di giochi in esame [Abr89]. Esso realizza una brutale ricerca su di un albero di gioco, nella quale è eseguito un esame esaustivo di tutte le possibili sequenze di mosse fino ai nodi terminali dell'albero. I risultati di tale visita sono fatti risalire (backing-up) lungo l'albero al fine di determinare un valore numerico (valore minimax) da associare al nodo radice. Il valore così ottenuto identifica la migliore mossa nella posizione cui il nodo radice si riferisce; in particolare è chiamata variante principale la sequenza di mosse dalla radice al nodo terminale lungo la quale è risalito il valore minimax del nodo radice.

L'algoritmo minimax poggia sull'ipotesi di gioco perfetto: in ogni posizione del gioco i giocatori selezionano sempre la mossa migliore secondo il loro punto di vista.

Assunto che MAX e MIN sono i nomi dei due giocatori, si intende ora descrivere formalmente come vengono assegnati i valori minimax ai nodi dell'albero di gioco.

I valori dei nodi foglia identificano l'esito finale del gioco dal punto di vista di MAX. Ai nodi non terminali, invece, i valori sono associati in maniera ricorsiva: il valore di un nodo in cui spetta a MAX la mossa è ottenuto massimizzando i valori dei suoi successori. Analogamente, se il nodo descrive una posizione in cui è MIN a dover muovere, allora il suo valore è il minimo dei valori dei successori (i valori sintetizzano sempre il punto di vista di MAX e quindi il suo antagonista cerca di minimizzarli nella loro risalita verso la radice).

Una misura della complessità degli algoritmi di ricerca su alberi di gioco è il numero dei nodi terminali visitati. In un albero di gioco uniforme con fattore di diramazione b e profondità d questo numero è fisso e pari a b^d.

Una variante della ricerca minimax è l'algoritmo negamax [KnuMoo75]. I due metodi sono essenzialmente gli stessi, ma nella ricerca negamax al livello superiore nell'albero di gioco è fatto risalire il valore negato dei suoi sottoalberi. In tale maniera i valori di un sottoalbero di un nodo sono sempre considerati dal punto di vista del giocatore cui spetta muovere in quel nodo. Pertanto l'algoritmo computa sempre il massimo di questi valori e non deve preoccuparsi di quale giocatore debba muovere.

In Fig. 1.3a e Fig. 1.3b sono raffigurati i due algoritmi.

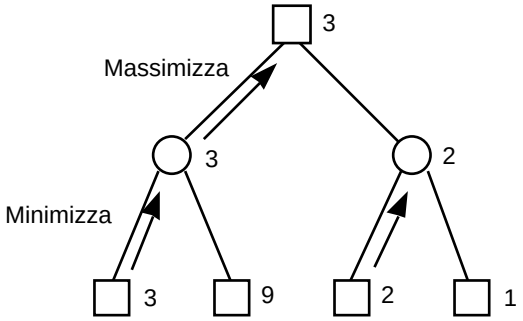


Fig. 1.3a Metodo minimax

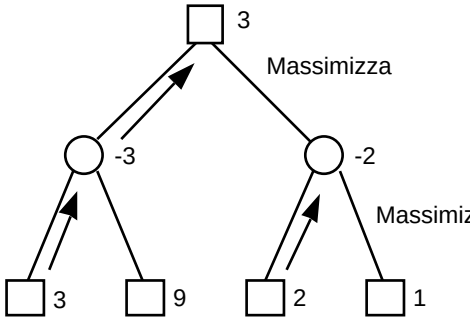


Fig. 1.3b Metodo negamax

La maggior parte dei giochi più interessanti generano alberi di gioco le cui dimensioni sono troppo grandi per consentire una ricerca completa; basti pensare che è stato stimato che il numero medio di nodi contenuti in un chess-tree (l'albero di gioco degli scacchi) è circa 10^{120} [dGr65].

In questi giochi la ricerca è troncata ad un certo stadio intermedio individuando così nell'albero una frontiera di ricerca, cioè un insieme di nodi interni (non terminali) non espandibili a causa di limiti computazionali.

A questi nodi, chiamati tip, è associato un insieme di informazioni sulle quali è applicata una funzione di valutazione statica, la cui natura è strettamente legata allo specifico dominio del gioco. Il valore di ritorno da tale funzione rappresenta una stima del valore effettivo del nodo tip, ovvero del suo valore minimax.

La suddetta funzione esegue pertanto una valutazione quantitativa; questa è il risultato dell'elaborazione di alcune espressioni aritmetiche legate ad alcune caratteristiche della posizione valutata [Cia92]. Normalmente essa ha struttura polinomiale, cioè scaturisce dalla combinazione lineare delle valutazioni separate di caratteristiche distinte della generica posizione p:

$$f(p) = \sum_i a_i \cdot t_i(p)$$

dove t_i individua una generica "sotto-valutazione" e a_i una misura della rispettiva importanza.

In molti giochi l'evidenza empirica mostra come l'inerente inaffidabilità della funzione di valutazione statica diminuisce con l'aumentare della profondità della ricerca [Abr89].

In Fig. 1.4 è presentato integralmente l'algoritmo minimax in versione negamax[3]. Le funzioni genmoves, makemove, undomove e evaluate sono dipendenti dal dominio: la prima determina i successori di una posizione, mentre le due seguenti gestiscono lo stato corrente del gioco eseguendo e retraendo l'esecuzione di una mossa; l'ultima, infine, è proprio la funzione di valutazione statica.

```
int minimax (position *p,int depth)
{
position *successor;
int nmoves, score, value, ind;
if (depth==0) /* raggiunta la profondità massima di ricerca?
*/
       return (evaluate(p));        /* è applicata la funzione
di valutazione statica */
nmoves=genmoves (p,&successor);     / *generazione mosse */
if (nmoves==0)        /* nodo terminale? */
       return (evaluate(p));
score=-∞;     /* inizializzazione del valore minimax
parziale */
/* ricerca del massimo score fra quello di tutti i
sottoalberi */
for (ind=0;ind<nmoves;ind++)
       {
       makemove (successor+ind);    /* esegui mossa */
       value=-minimax(successor+ind,depth-1);
       if (value>score)
              score=value;
       undomove (successor+ind);    /* retrai la mossa
valutata */
       }
return (score);
}
```

Fig. 1.4 Algoritmo minimax (versione negamax)

1.2.2 La funzione di valutazione, l'effetto orizzonte e la ricerca quiescente

Quando la ricerca raggiunge un nodo terminale di un albero di gioco è invocata una funzione di valutazione statica per assegnare un valore alla posizione corrente. Tale funzione non tiene conto di quanto potrà accadere nelle mosse successive allo stato del gioco esaminato. L'insieme di questi nodi foglia, tutti posti ad una fissata profondità, costituisce un orizzonte al di là del quale la funzione di valutazione non ha visibilità. Questo fatto può essere sorgente di gravi errori di valutazione dovuti appunto a questo fenomeno conosciuto come effetto orizzonte [Ber73].

Vi sono posizioni che più di altre possono essere fonte di errore e la loro caratterizzazione dipende fortemente dal gioco in esame. Esse sono chiamate instabili.

Ad esempio negli scacchi sono considerate instabili posizioni che occorrono dopo la cattura di un pezzo o dopo uno scacco al re.

Si cerca di diminuire l'effetto orizzonte estendendo di qualche mossa la ricerca oltre i nodi foglia considerati instabili: questa estensione è chiamata ricerca quiescente.

La ricerca quiescente può comportare un sensibile aumento della complessità computazionale. Per limitare questo overhead si restringe l'esplorazione oltre il nodo terminale ad un sottoinsieme delle mosse legali. Ad esempio, se la cattura di un pezzo ha originato una posizione instabile,

allora durante la ricerca quiescente sono analizzate solamente le mosse di cattura, contenendo così la crescita delle dimensioni dell'albero di gioco.

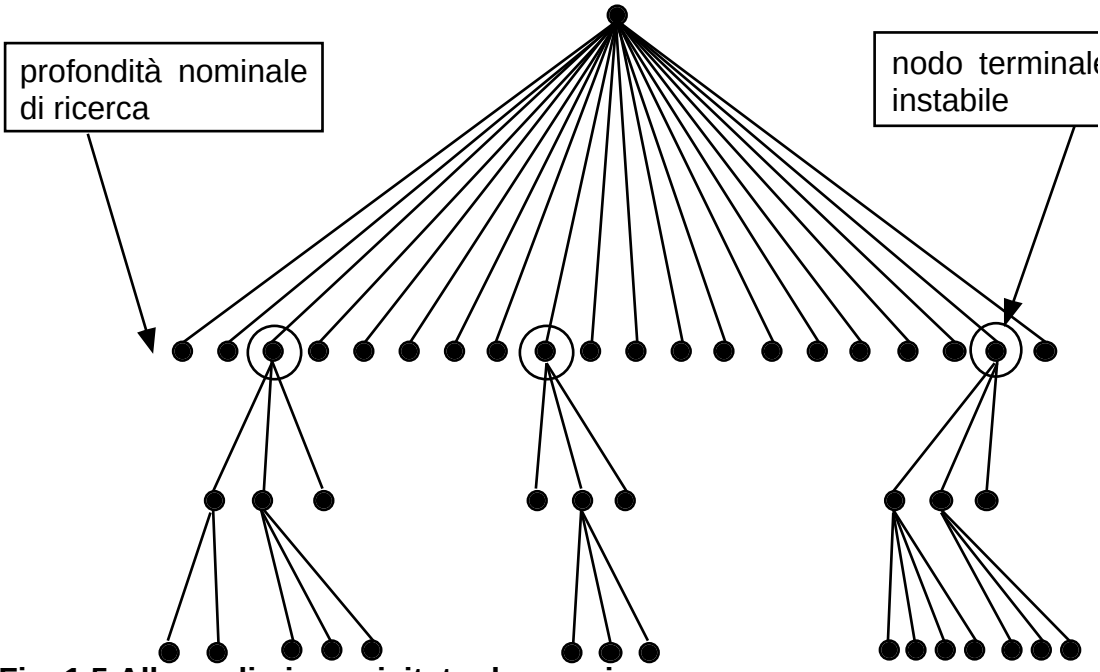


Fig. 1.5 Albero di gioco visitato da una ricerca quiescente

In Fig. 1.5 è illustrato l'albero di gioco esplorato dall'algoritmo minimax in combinazione con la ricerca quiescente: si osservi come solo alcune linee di gioco si estendano oltre la profondità massima fissata dall'algoritmo minimax (denominata profondità nominale di ricerca).

1.2.3 Ordine di visita dell'albero di gioco: discesa a scandaglio (depth-first search)

Accanto alle regole per far risalire il valore minimax verso la radice è importante definire l'ordine di visita dell'albero di gioco.

Il calcolo del valore minimax di un nodo è eseguito generalmente con una visita a scandaglio (depth-first search) realizzata con una forma di backtracking.

I numeri con cui sono etichettati i nodi di Fig. 1.6 mostrano l'ordine di generazione degli stessi nodi in una visita a scandaglio. Il vantaggio principale di questo ordine di ricerca è l'estrema semplicità della procedura di gestione (facilmente modellabile tramite ricorsione) ed il modesto spazio di memoria richiesto. Quest'ultimo è infatti lineare con la lunghezza del cammino corrente dalla radice e contiene una struttura a pila composta dei nodi di questo cammino.

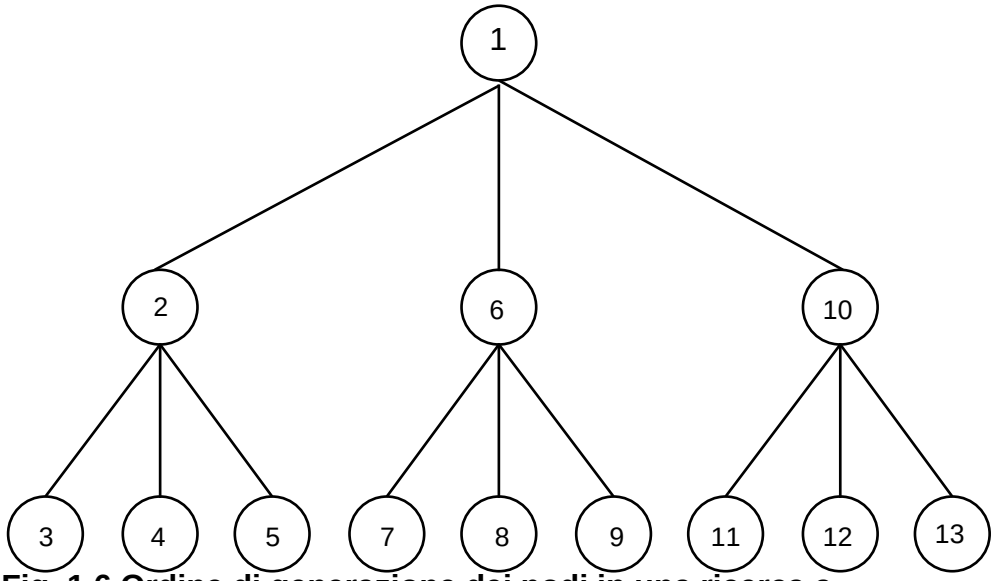


Fig. 1.6 Ordine di generazione dei nodi in una ricerca a scandaglio

1.3 Algoritmi di ricerca sequenziale

Data una posizione p in un gioco a somma zero di 2 giocatori e fissato un albero di gioco che rappresenti tutte le possibili continuazioni del gioco a partire da p, lo scopo di un algoritmo di ricerca è quello di determinare il valore minimax del nodo p. Questo valore sarà una approssimazione di quello reale nel caso che l'albero visitato sia parziale, cioè i valori associati ai suoi nodi tip non siano esatti, ma frutto di una stima.

L'algoritmo minimax ottiene correttamente questa finalità, ma ad un costo computazionale insoddisfacente.

A partire dal 1950 un'importante osservazione ha determinato lo sviluppo di una tecnica di ricerca che ora è uno standard: esiste un insieme di mosse facilmente riconoscibile che non saranno selezionate dall'algoritmo minimax.

Il metodo che poggia su tale considerazione si chiama $\alpha\beta$-pruning.

1.3.1 L'algoritmo $\alpha\beta$

L'algoritmo $\alpha\beta$ produce lo stesso risultato della ricerca minimax, ma ad un minor costo. Esso, infatti, effettua dei tagli (cutoff) lungo l'albero di gioco, ovvero determina sottoalberi che non contengono il valore minimax ottimo per il nodo radice e quindi non procede alla loro visita. L'esatta origine dell'algoritmo $\alpha\beta$ non è nota; i primi documenti in cui è discusso in dettaglio sono [EdwHar63] e [Bru63].

L'algoritmo opera tenendo traccia dei limiti dell'intervallo (α,β) all'interno del quale deve essere contenuto il valore minimax di un nodo. L'intervallo (α,β) è chiamato finestra $\alpha\beta$ ($\alpha\beta$-window).

In particolare il parametro α è un limite inferiore per il valore che deve essere assegnato ad un nodo che massimizza,

mentre β è un limite superiore per il valore di un nodo che minimizza sui valori dei successori. Sono "recisi" i nodi il cui valore cade al di fuori dell'intervallo (α,β) ed i rispettivi sottoalberi non ancora visitati sono quindi ignorati dalla ricerca.

Gli esempi che seguono illustrano una giustificazione intuitiva del metodo ed una classificazione dei tagli [Cia92; KnuMoo75].

Si consideri l'albero di gioco di Fig. 1.7. La prima mossa valutata dal bianco è ♙g2-g3 il cui valore minimax è $v > -\infty$. La successiva mossa valutata è ♕h5-f7; in risposta a tale mossa è analizzata la mossa ♖a3-a1 la quale origina immediatamente il matto per il nero: indipendentemente dalle altre mosse di risposta del nero il valore minimax finale per la mossa ♕h5-f7 equivale ad una sconfitta ($=-\infty$) e quindi questa non è una scelta conveniente per il bianco.

Nella situazione descritta, pertanto, la porzione del sottoalbero inesplorato relativa a ♕h5-f7 è inutile che venga visitata ed è quindi "tagliata" dalla ricerca. In particolare si dice che la mossa ♕h5-f7 è stata confutata (rifiutata) dalla mossa ♖a3-a1, cioè questa rappresenta una risposta ad essa sufficiente a dimostrarne l'inopportunità.

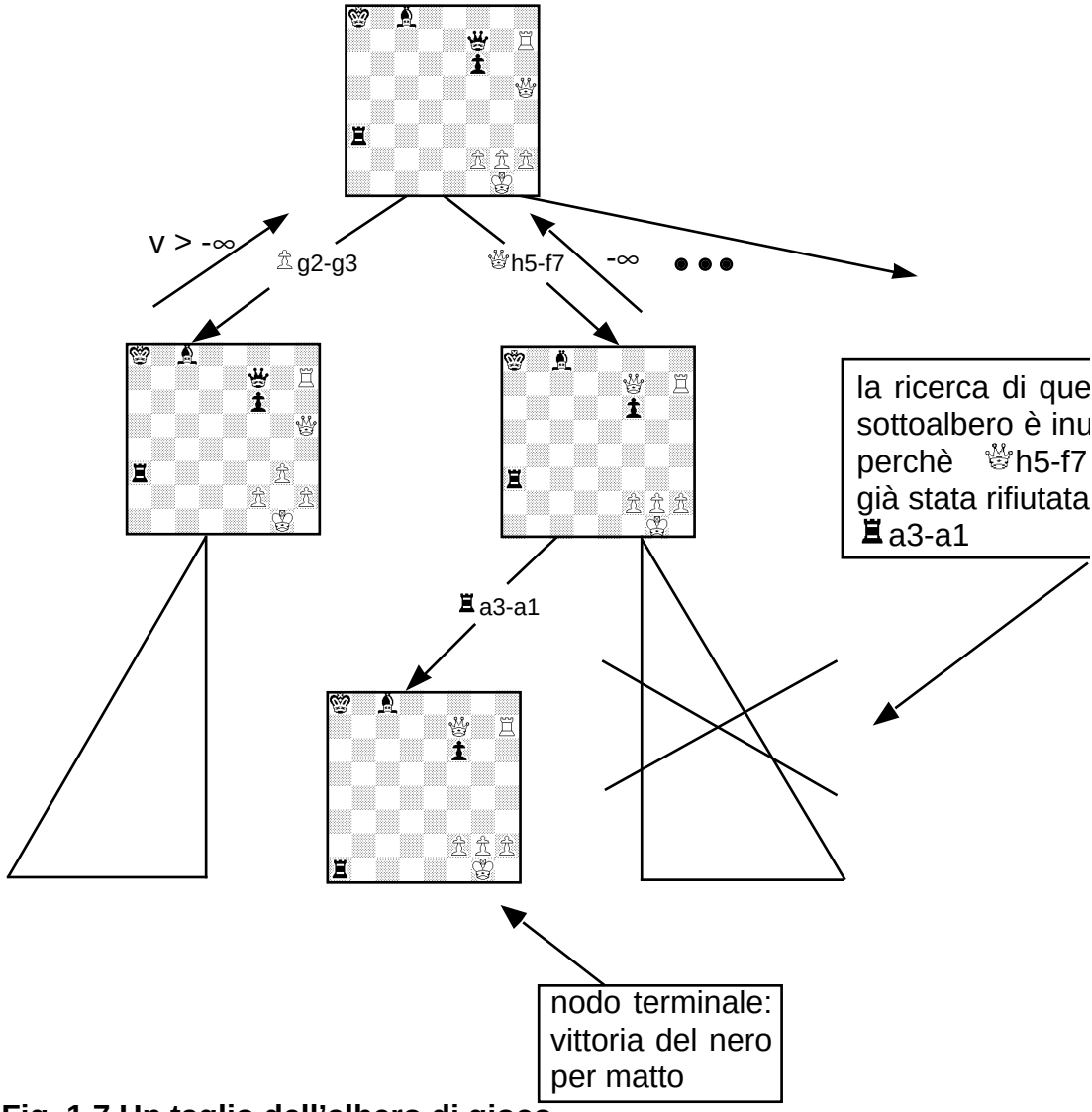


Fig. 1.7 Un taglio dell'albero di gioco

I tagli eseguiti dall'algoritmo $\alpha\beta$ possono essere di due tipi: in superficie o in profondità.

Si considerino a riguardo le due porzioni di albero di gioco in Fig. 1.8a e 1.8b.

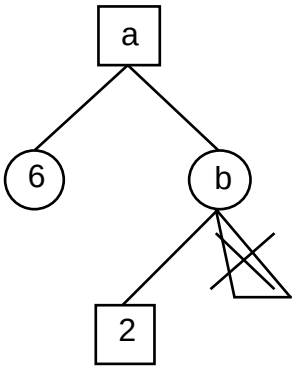
In entrambi gli alberi, per alcuni nodi è stato calcolato il valore finale, mentre gli altri, etichettati con una lettera, attendono che il valore minimax sia loro assegnato.

Il valore finale per il nodo radice a in Fig. 1.8a è dato da:

a=max(6,b) dove b=min(2,…).

È chiaro che a=6, indipendentemente dal valore esatto di b: avviene un taglio denominato taglio in superficie.

Analogamente, in Fig. 1.8b il valore della radice a può essere ottenuto senza conoscere l'esatto valore del nodo d. In questo caso si parla di taglio in profondità poiché le informazioni ricavate ad un certo livello hanno provocato un taglio ad una profondità maggiore.



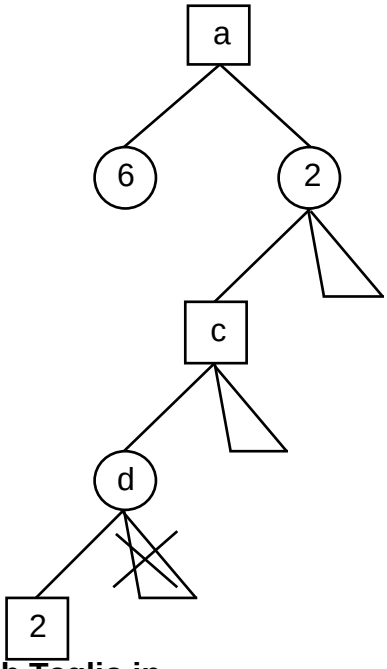


Fig. 1.8a Taglio in superficie **Fig. 1.8b Taglio in profondità**

Al fine di assicurare che sia trovato il corretto valore minimax della radice i parametri α e β sono inizializzati rispettivamente a $-\infty$ e $+\infty$. Essi saranno poi aggiornati durante la visita dell'albero di gioco.

Normalmente l'utilizzo dell'algoritmo $\alpha\beta$ è stabilito da una chiamata di funzione della forma:

`MM=alphabeta(p,alpha,beta,depth)`
dove p è un puntatore ad una struttura che descrive lo stato del gioco nel nodo esplorato; (alpha,beta) è la finestra $\alpha\beta$ e depth la lunghezza massima del cammino di ricerca espressa come numero di semi-mosse (ply). Il valore MM ritornato dalla funzione è il valore minimax per la posizione p.

Fig. 1.9 illustra una versione negamax dell'algoritmo $\alpha\beta$.

La versione dell'algoritmo proposta è molto semplice e dovrebbe essere estesa per tenere traccia della mossa ottimale trovata.

```
int alphabeta(position *p,int alpha,int beta,int depth)
{
position *successor;
int nmoves,score,value,ind;
if (depth==0)
       return (evaluate(p));
nmoves=genmoves(p,&successor);
if (nmoves==0)
       return (evaluate(p));
score=alpha;  /* il valore minimax parziale ha valore
iniziale alpha e non -∞ */
```

```
for (ind=0;i<nmoves;ind++)
       {
       makemove(successor+ind);
       value=-alphabeta(successor+ind,-beta,-score,depth-1);
       undomove(successor+ind);
       if (value>score)      /* il valore calcolato fornisce
un miglioramento? */
              score=value;
       if (score>=beta)      /* un taglio? */
              return(score);        /* la visita del restante
sottoalbero è "recisa" */
       }
return (score);
}
```

Fig. 1.9 Algoritmo αβ (versione negamax)

1.3.1.1 L'efficienza dell'algoritmo αβ

Knuth e Moore hanno ridefinito la struttura di un albero di gioco quando è oggetto di una ricerca αβ [KnuMoo75]. Essi hanno classificato i nodi che lo compongono in tre categorie:

• nodi PV: sono i nodi che compongono la continuazione principale (principal variation), cioè il primo cammino dalla radice ad un nodo terminale che viene esaminato nella ricerca[4].

• nodi CUT: rappresentano alternative dei nodi PV. Sono gli unici nodi in cui possono avvenire tagli.

• nodi ALL: sono nodi successori dei nodi CUT. A loro volta i successori dei nodi ALL sono ancora di tipo CUT.

Mentre tutti i sottoalberi dei nodi PV e ALL sono sempre esplorati, solamente un sottoinsieme dei sottoalberi dei nodi CUT sarà preso in considerazione a causa dei tagli.

Fig. 1.10 illustra la struttura presentata.

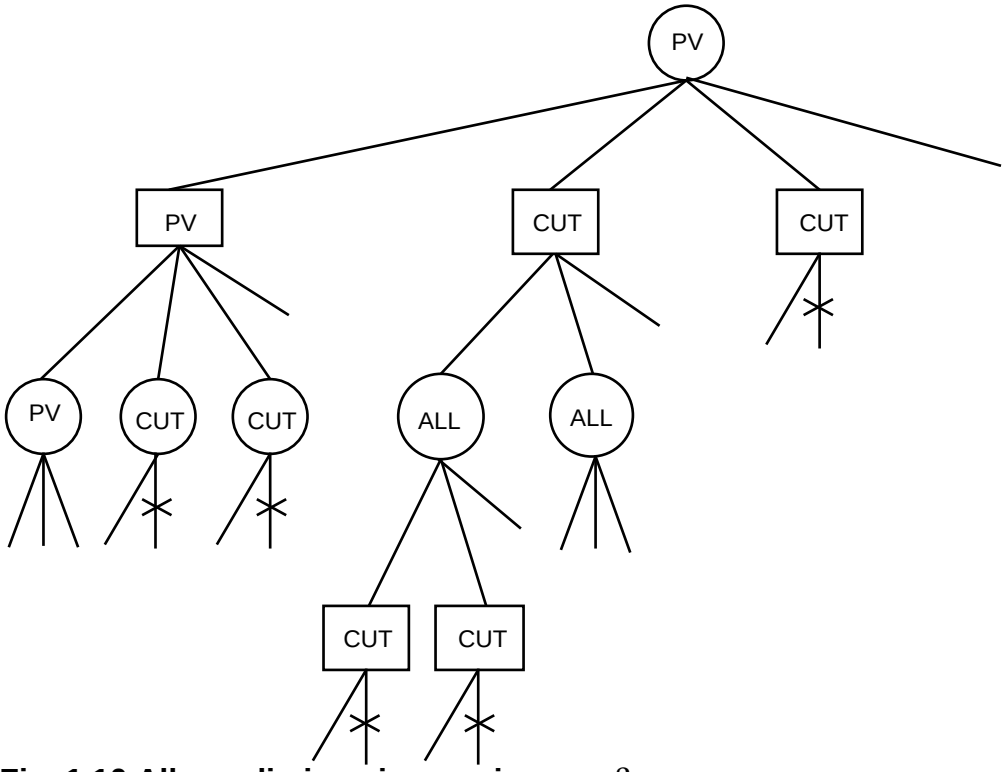


Fig. 1.10 Albero di gioco in una ricerca $\alpha\beta$

Nel caso ottimale di ricerca $\alpha\beta$ il primo discendente diretto di ogni nodo CUT determina un taglio. L'albero esplorato in questa situazione speciale viene chiamato albero minimale. Esso definisce la porzione minima dell'albero di gioco che verrà comunque esplorata da un algoritmo $\alpha\beta$.

Dovendo confrontare le prestazioni degli algoritmi minimax e $\alpha\beta$ si considera come riferimento la ricerca su un albero di gioco uniforme di fattore di diramazione b e profondità d. È anche usuale confrontare i due metodi in termini del numero di nodi terminali valutati.

Alphabeta riduce le dimensioni dell'albero di gioco, ma non ne elimina la crescita esponenziale. Nel caso ottimo il numero n_t di nodi terminali che devono essere valutati è:

$$n_t = \begin{cases} 2 \cdot w^{d/2} - 1 & \text{se d è pari} \\ w^{(d+1)/2} + w^{(d-1)/2} & \text{se d è dispari} \end{cases}$$

dove w è il fattore di diramazione e d la profondità di ricerca [KnuMoo75].

Nel caso pessimo alphabeta diviene minimax e valuta pertanto w^d nodi terminali.

Pearl ha dimostrato che per alberi con attribuzione casuale dei valori ai nodi terminali, nel tempo che minimax completa una ricerca a profondità d,

alphabeta esplora in media un albero di profondità $\frac{4}{3}d$ [Pea82].

Va sottolineato che la prestazione ottimale è ottenuta solo quando la prima mossa considerata ad ogni nodo è la migliore. In questo caso l'albero si dice perfettamente ordinato. In generale, un albero di gioco è ordinato se la mossa migliore in un generico nodo è fra le prime ad essere esaminata.

Il caso pessimo, invece, si crea quando le mosse sono ordinate in ordine inverso: dalla peggiore alla migliore.

L'efficienza dell'algoritmo $\alpha\beta$ è tanto più grande quanto più forte è l'ordinamento dell'albero [SlaDix69]. Poiché la differenza fra le dimensioni dell'albero minimale e massimale sono notevoli, è imperativo ottenere un buon ordinamento dei nodi interni; in alcuni dei paragrafi successivi sarà estesa la discussione di questo problema.

1.3.1.2 Una caratterizzazione dell'albero minimale

L'albero minimale è quello esplorato dall'algoritmo $\alpha\beta$ in condizioni ottime di ordinamento dell'albero di gioco.

È possibile dare una definizione più formale di albero minimale.

Per semplificare l'esposizione è introdotta la seguente terminologia: il primo arco uscente da un nodo punta al "figlio sinistro", mentre gli altri puntano ai figli "destri".

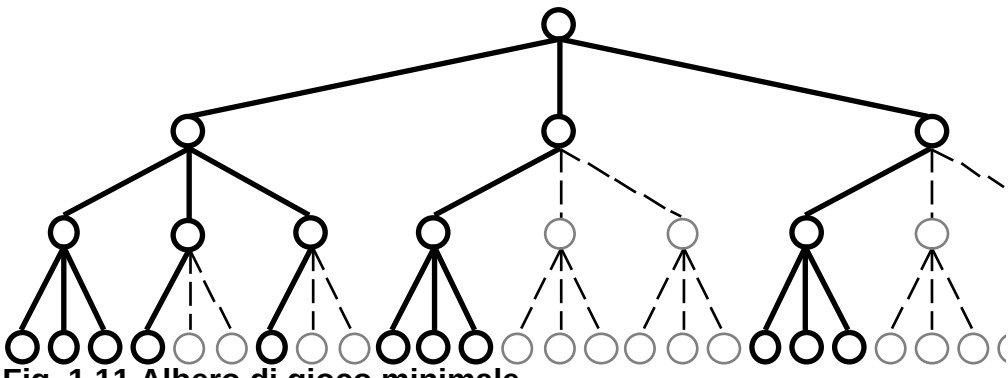


Fig. 1.11 Albero di gioco minimale

La struttura dell'albero minimale è definita ricorsivamente dalle regole:

• il nodo radice fa parte dell'albero minimale

• il figlio sinistro di un nodo dell'albero minimale appartiene esso stesso all'albero minimale

• il figlio sinistro di ogni figlio destro di un nodo dell'albero minimale fa parte anch'esso dell'albero minimale.

In Fig. 1.11 è evidenziato l'albero minimale di un albero di gioco uniforme di profondità 3 e fattore di diramazione 3.

1.3.2 Alcune varianti dell'algoritmo αβ

1.3.2.1 Aspiration search

L'efficienza di un algoritmo αβ è strettamente legata al numero di tagli che riesce a produrre. La probabilità di operare un taglio nella ricerca di un albero di gioco è tanto maggiore quanto più stringente è l'intervallo fra i parametri α e β.

In alcune applicazioni, fra le quali gli scacchi, esistono metodi per stimare con buona approssimazione il valore minimax di una posizione. A partire da questa stima V e fissato un margine di errore err, si origina una ricerca αβ con una finestra iniziale (V-err,V+err) in sostituzione della finestra canonica (-∞,+∞). Questa finestra più stringente (detta aspiration window) "aspira" a contenere il valore minimax corretto, da cui il nome del metodo: aspiration search [MarCam82].

```
int aspiration_search (position *p,int v,int err,int
depth)
{
int alpha,beta,value;
alpha=v-err;
beta=v+err;
value = alphabeta(p,alpha,beta,depth);
if (value>=beta)      /* fallimento superiore? */
       value=alphabeta (p,beta,+∞,depth)
else if (v<=alpha)    /* fallimento inferiore? */
       value=alphabeta(p,-∞,alpha,depth);
return(value);
}
```

Fig. 1.12 Aspiration search

Naturalmente esiste il rischio che il valore stimato risulti errato ed il valore minimax cada al di fuori della aspiration window. In tale evenienza la ricerca deve essere ripetuta con una differente finestra di ricerca.

In Fig. 1.12 è descritto l'algoritmo aspiration search.

1.3.2.2 Falphabeta (Fail-soft Alphabeta)

Falphabeta [FisFin80;MarCam82] è una variante dell'algoritmo αβ molto utile quando impiegata in combinazione con aspiration search.

L’algoritmo è ottenuto apportando due modifiche alla funzione alphabeta di Fig. 1.9:

• la variabile score è inizializzata a -∞ invece che al valore di alpha

• l’invocazione ricorsiva diventa:

```
value=-alphabeta(successor,-beta,-
max(alpha,score),depth-1);
```

Quando l’algoritmo falphabeta è usato in aspiration search fornisce un limite più stringente per il valore corretto se la prima ricerca fallisce (visitando lo stesso numero di nodi dell’algoritmo αβ).

1.3.2.3 PVS (Principal Variation Search) ed il concetto di finestra minimale

Palphabeta [FisFin80] e Scout [Pea80] sono due algoritmi che hanno introdotto un approccio differente al calcolo efficiente dell’esatto valore minimax. L’idea alla base di questi algoritmi è l’assunzione che la prima mossa esaminata in ogni posizione sia la migliore. Emerge quindi un insieme di mosse che definisce un cammino che rappresenta l’ipotetica continuazione principale.

Gli algoritmi operano inizialmente una ricerca lungo questo cammino associando un valore statico al nodo foglia raggiunto. Successivamente si passa a verificare se le mosse alternative a quelle del cammino esaminato apportano o meno valutazioni minimax migliori. Questo tipo di test è meno costoso della computazione del valore minimax per tutti i sottoalberi relativi. Tuttavia, qualora una mossa alternativa origini un valore migliore, essa definisce la nuova continuazione principale corrente ed è necessario operare una nuova ricerca nel suo sottoalbero per conoscere il suo esatto valore minimax.

L’operazione di test è spesso realizzata facendo uso del concetto di finestra minimale (minimal window): una finestra di tipo αβ della forma (V,V+1).

Tale finestra è detta minimale perché non contiene alcun valore. Fig. 1.13 mostra un esempio di algoritmo di ricerca con finestra minimale.

Mentre la prima mossa è valutata in maniera ricorsiva, l’esame delle mosse alternative prevede che inzialmente sia invocata una ricerca falphabeta con minimal window (V,V+1), dove V è il miglior valore minimax trovato fino a quel momento. Lo scopo di tale chiamata è proprio quello di controllare se l’alternativa ha un valore migliore. Ogni volta che tale condizione si verifica è eseguita ancora una

ricerca falphabeta con una finestra più estesa al fine di computare il valore esatto.

```
int mws(position *p)
{
position *successor;
int nmoves,score,value,ind;
nmoves=genmoves(p,&successor);
if (nmoves==0)
       return(evaluate(p));
score=-mws(successor);       /* valutazione lungo la
variante principale */
for (ind=1;i<nmoves;ind++)
       {
       value=-falphabeta(successor+ind,-score-1,-
score);
       if (value>score)      /* l'alternativa contiene
un valore migliore? */
              score=-falphabeta(successor+ind,-∞,-
value);
       }
return(score);
}
```

Fig. 1.13 Ricerca con finestra minimale

PVS (principal variation search) è un esempio di applicazione all'algoritmo αβ del concetto di ricerca con finestra minimale [MarCam82]. L'algoritmo, descritto in Fig. 1.14, estende infatti questa ricerca con la gestione di una finestra (α,β) al fine di aumentare il numero di tagli prodotti durante la visita dell'albero di gioco e rendere così possibile anche una combinazione del metodo con una forma di aspiration search.

```
int pvs(position *p,int alpha,int beta,int depth)
{
position *successor;
int nmoves,score,value,ind;
if (depth==0)
       return(evaluate(p));
nmoves=genmoves(p,&successor);
if (nmoves==0)
       return(evaluate(p));
best=-pvs(successor,-beta,-alpha,depth-1);
for (ind=1;i<nmoves;ind++)
       {
       if (best>=beta)
              return(score);
       alpha=max(score,alpha);
       value=-pvs(successor+ind,-alpha-1,-
alpha,depth-1);
       if ((value>alpha) && (value<beta))
              score=-pvs(successor,-beta,-
value,depth-1);
       else if (value>score)
              score=value;
       }
return(score);
}
```

Fig. 1.14 PVS

Va sottolineata la forte dipendenza dell'efficienza di PVS rispetto al più o meno forte ordinamento dell'albero. Infatti un albero di gioco non ordinato contiene molti nodi interni il cui valore minimax è migliore dei fratelli più anziani e ciò implica una duplice visita del sottoalbero ad essi associato.

Esiste quindi la possibilità che una ricerca PVS esamini più nodi di un algoritmo $\alpha\beta$. I vantaggi computazionali di PVS emergono quando è realizzato in combinazione con delle euristiche che stabiliscono un ordinamento nella generazione delle mosse, tale da aumentare la probabilità che le mosse valutate per prime siano le migliori.

1.3.3 Euristiche per migliorare la ricerca $\alpha\beta$

L'algoritmo $\alpha\beta$ è divenuto uno standard di ricerca grazie alla sua capacità di trovare sempre la combinazione di mosse migliore pur esaminando un minor numero di mosse rispetto a minimax.

È stato spiegato come le prestazioni di questo algoritmo siano fortemente dipendenti dall'ordinamento dei nodi interni dell'albero: se la migliore mossa non è esplorata per prima, allora è necessario un lavoro addizionale per verificare che un'altra mossa è migliore di quella presunta ottima. Ciò può accadere più volte in una ricerca: il caso pessimo occorre quando le mosse sono esplorate in ordine inverso rispetto al loro effetivo valore.

Nella pratica l'albero visitato non è mai quello minimale. Poiché le dimensioni dell'albero sono fortemente influenzate dall'ordine con cui le mosse sono visitate, è importante che la mossa migliore sia esaminata al più presto. Esaminare questa mossa per prima non è sempre possibile: ciò infatti starebbe a significare che la migliore mossa è nota a priori e quindi non si avrebbe bisogno di alcuna ricerca. Sono stati sviluppati metodi che, con un elevato grado di probabilità, sono in grado di predire la mossa migliore, permettendo così di visitarla per prima [MarCam82].

In seguito sono presentate alcune euristiche, definite di ricerca, il cui scopo è di migliorare la ricerca $\alpha\beta$ favorendo la minimizzazione delle dimensioni dell'albero visitato.

1.3.3.1 Approfondimento iterativo (Iterative deepening)

A differenza degli algoritmi finora visti che sono specifici per i giochi a somma zero di 2 giocatori, l'approfondimento iterativo è un paradigma di ricerca assolutamente generale.

Inserito nel contesto dei metodi αβ, esso definisce un processo iterativo nel quale si fa uso della ricerca con profondità nominale (d-1) per preparare la successiva ricerca a profondità d. In altre parole viene prima eseguita una visita completa dei nodi a profondità 1, poi a profondità 2, a profondità 3 e così via.

I vantaggi offerti da tale tecnica sono molti:

• limitazione in tempo della ricerca: fissata una soglia limite per il tempo totale di ricerca, questa procede a profondità massime successive fino a che il limite temporale non è raggiunto.

• controllo della ricerca: in alcune applicazioni, ad esempio l'analisi di certi finali negli scacchi, la soluzione può essere trovata dopo un numero esiguo di mosse. In tale contesto l'algoritmo di ricerca non conosce a quale profondità nell'albero di gioco incontrerà la soluzione. Vi sarebbe un'enorme perdita di efficienza se un problema risolvibile in una mossa fosse analizzato da una ricerca a profondità 5!

La tecnica di approfondimento iterativo elimina tale anomalia garantendo che un problema di N mosse sia risolto da una ricerca di profondità al più N.

• ordinamento delle mosse: dopo ogni iterazione la lista delle mosse legali nel nodo radice viene riordinata in maniera che le mosse risultate migliori in una data ricerca siano esaminte per prime nella successiva iterazione. In particolare, la continuazione principale trovata nella iterazione (d-1) rappresenta la sequenza iniziale di mosse esplorate nella d_esima ricerca.

• miglioramento di aspiration search: il valore minimax finale di ritorno dalla ricerca a profondità (d-1) può essere usato come centro di una aspiration window nella iterazione successiva. È molto probabile che questa finestra contenga il valore minimax della nuova ricerca.

Il maggiore vantaggio di iterative deepening è però il

• riempimento delle tabelle: diverse euristiche che saranno introdotte tra breve fanno riferimento a tabelle in cui sono conservate informazioni del tipo: la valutazione di certe posizioni oppure quali mosse hanno prodotto un taglio.

Il metodo di approfondimento iterativo favorisce, ad esempio, il frequente ripresentarsi di posizioni già analizzate e rende pertanto molto significativo l'utilizzo di queste tavole. I vantaggi di questa tecnica furono dimostrati per primo da Chess 4.5 [SlaAtk77],

uno dei più potenti programmi di scacchi degli anni ’70 e poi provati formalmente da Korf [Kor85].

Esiste una forma alternativa di approfondimento iterativo, in cui la successiva iterazione prevede che la ricerca sia condotta ad una profondità massima aumentata di 2 rispetto all’iterazione corrente. Tale metodologia è giustificata formalmente da Nau in [Nau82]. Egli ha dimostrato la tendenza della ricerca minimax ha restituire valutazioni vincenti per il giocatore quando essa è condotta a profondità dispari e perdenti se a profondità pari. I valori ritornati da ricerche a profondità alterne formano dunque due sequenze distinte che devono essere considerate separatamente. È quindi evidente lo scopo in questa tecnica di preservare continuità fra iterazioni successive, evitando l’alternarsi di linee di gioco difensive e di attacco.

```
int iterative_deepening (position *p,int maxdepth)
{
int value, d;
for (d=1;d<=maxdepth;d++)
       {
       value=alphabeta(p,-∞,+∞,d);
       sort(p);      /* le mosse sono riordinate in
modo che la migliore sia visitata
       per prima nella successiva iterazione */
       }
return(value);
}
```

Fig. 1.15 Iterative deepening

In Fig. 1.15 è illustrato un algoritmo di ricerca che implementa la tecnica di approfondimento iterativo insieme ad un meccanismo di ordinamento nella generazione delle mosse al livello della radice (top-level).

1.3.3.2 Tabella delle trasposizioni (Transposition Table)

Durante la ricerca di un albero di gioco il medesimo stato del gioco può presentarsi più volte. La ragione di questo fenomeno è la presenza di trasposizioni: lo stesso stato del gioco è raggiunto da sequenze di mosse differenti.

Invece di valutare nuovamente queste posizioni ricorrenti si recuperano da una tabella le informazioni ottenute da una valutazione precedente dello stesso nodo.

Ogni riga di tale tabella, detta appunto delle trasposizioni [MarCam82], rappresenta una posizione

del gioco. Tali righe sono gestite con un'organizzazione hash per favorirne un breve tempo di ricerca. È quindi definita una funzione hash h: stato→int che associa ad ogni posizione una chiave intera utilizzata per indicizzare la tabella.

I campi che normalmente compongono un in elemento della tabella delle trasposizioni sono i seguenti:

- lock
- giocatore
- lunghezza
- valore
- flag

L'esigenza del campo lock è legata alla possibilità che la funzione hash non sia uniforme, cioè che posizioni differenti siano mappate nella stessa chiave:

$h(s_i)=h(s_j)$ e $s_i \neq s_j$

Il campo lock contiene l'identificatore della posizione rappresentata nella riga. Qual è il suo utilizzo?

Si supponga che debba essere analizzato il sottoalbero relativo alla posizione X. La funzione hash determina che R=h(X) è la riga corrispondente ad X nella tabella.

Si consideri l'ipotesi che nella riga R sia già memorizzata la posizione Y. Viene allora confrontato il campo lock della riga R con l'identificatore di X: se essi non coincidono si deve valutare la posizione X in quanto nessuna informazione in suo merito è contenuta nella tabella.

Il risultato della valutazione di X verrà sostituito a quello di Y nella riga R solo se la profondità della valutazione di X è maggiore di quella di Y (registrata nel campo lunghezza di R). In tale modo viene conservata l'informazione più costosa in termini computazionali.

Il campo giocatore indica il punto di vista secondo il quale la valutazione è avvenuta. Il valore di una posizione valutata da un giocatore è diverso, in generale, dalla valutazione negata effettuata dall'avversario nella stessa posizione. Infatti il calcolo del valore minimax è influenzato dal fatto che il giocatore corrente sia in attacco o in difesa, inteso che è considerato in attacco il giocatore cui spetta la mossa nel nodo radice.

Il campo lunghezza indica la profondità del sottoalbero valutato. Normalmente la tabella è riempita con valutazioni la cui profondità deve essere superiore ad un certo valore di soglia; questa scelta è

dovuta al fatto che la memorizzazione e la lettura di informazioni relative alla valutazione di piccoli sottoalberi è più costosa della sua nuova valutazione a causa dell'overhead introdotto dalla gestione della tabella.

Il campo valore contiene la valutazione calcolata per la posizione. Questa è il valore minimax oppure una sua limitazione superiore o inferiore a secondo di quanto specificato nel campo flag. Deve essere infatti chiarito che l'algoritmo $\alpha\beta$ calcola il valore minimax di un nodo solo se quest'ultimo cade nella finesta (α,β) iniziale. In generale il valore v(p) restituito dall'algoritmo è nella seguente relazione con l'esatto valore minimax mm(p) della posizione:

- $v(p) \leq \alpha \Rightarrow v(p) > mm(p)$ (il fallimento inferiore fornisce una limitazione superiore)
- $\alpha < v(p) < \beta \Rightarrow v(p) = mm(p)$ (il valore calcolato è quello esatto)
- $v(p) \geq \beta \Rightarrow v(p) < mm(p)$ (il fallimento superiore fornisce una limitazione inferiore)

Si osservi la tabella delle trasposizioni al lavoro: se una posizione raggiunta durante la ricerca è già contenuta nella tabella ed il campo giocatore coincide con quello corrente, allora si controlla se il valore nel campo lunghezza è $\geq$ alla rimanente profondità di ricerca per la posizione corrente. Se tale test ha successo, i contenuti dei campi valore e flag possono essere usati per accelerare o sopprimere la valutazione della posizione. In caso contrario la posizione deve essere valutata nuovamente poiché le informazioni nella tabella riguardano un suo sottoalbero non sufficientemente esteso.

Tale euristica restringe le dimensioni dell'albero di gioco. La riduzione è maggiore in situazioni di gioco in cui è alta la probabilità di generare trasposizioni, ad esempio nei finali degli scacchi dove si hanno pochi pezzi ed un elevato numero di mosse reversibili.

1.3.3.3 Euristica dei killer

Viene denominata killer una mossa che ha prodotto un taglio durante la ricerca $\alpha\beta$ di un dato sottoalbero [Cia92;MarCam82]. Tale tecnica è basata sull'intuizione che una mossa killer possa produrre ulteriori tagli nell'analisi di altri sottoalberi.

Si consideri a riguardo la situazione scacchistica di Fig. 1.16 in cui la mossa spetta al nero. Molte delle mosse del nero (ad esempio ♝a7-a6) sono facilmente confutate dalla minaccia al re portata dal

bianco: ♕h4xh7; vi sono infatti poche mosse che prevengono tale minaccia. L'euristica dei killer prevede che la mossa ♕h4xh7 sia ricordata per le sua proprietà di avere confutato almeno una mossa del nero: nell'analisi delle successive mosse del nero essa sarà esaminata come prima mossa di risposta nella speranza di ottenere una loro rapida confutazione.

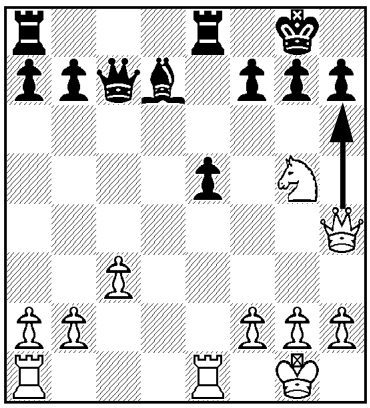


Fig. 1.16 Euristica dei killer

Normalmente per ogni livello dell'albero di gioco viene gestita una breve lista di mosse killer. Generando per prime le mosse killer si cerca quindi di massimizzare la probabilità che una linea di gioco che sarà origine di un taglio sia recisa molto presto risparmiando così tempo di ricerca.

L'euristica dei killer può comportare un riordinamento delle mosse sia dinamico che statico.

Nella prima situazione il riordinamento è fatto nell'ambito della stessa visita dell'albero di gioco: una mossa killer che ha prodoto un taglio al livello n potrebbe essere tentata ai livelli n, n-2, n-4, … (sempre che qui sia legale) prima che siano generate le restanti alternative.

L'ordinamento statico delle mosse è invece effettuato tra una iterazione e la successiva in una metodologia di approfondimento iterativo.

1.3.3.4 Tabella delle confutazioni (o delle refutazioni)

Il maggiore svantaggio della tabella delle trasposizioni è la sua dimensione. La tabella delle confutazioni intende conservare i principali giovamenti della tabella delle trasposizioni, ma richiedendo una minore occupazione di memoria [Sch86].

Questa euristica è introdotta in combinazione con quella di approfondimento iterativo. Dopo ogni ricerca a profondità d, sono memorizzate in una tabella (tavola delle confutazioni) certe sequenze di mosse dal nodo radice al nodo foglia.

Ogni riga della tabella rappresenta una mossa legale nella radice. Per la mossa risultata migliore dopo l'ultima iterazione la tabella contiene la continuazione principale, mentre per le sue mosse alternative memorizza una sequenza di al più d mosse sufficienti a confutare queste mosse. Nella successiva ricerca a profondità (d+1) la tabella è utilizzata per guidare ogni mossa lungo una potenziale linea di confutazione riducendo così sensibilmente il tempo di computazione.

La tabella delle confutazioni deve memorizzare la continuazione principale per ognuna delle mosse candidate al top-level. Le sue dimensioni sono quindi:

$max_w \cdot max_d$

dove max_w è il numero massimo di discendenti della radice (per gli scacchi tipicamente 35, ma, nel caso peggiore, circa 90) e max_d è la massima profondità di ricerca [MarPop85].

1.3.3.5 Tabella history

Anche la tabella history [FeMyMo90] rappresenta un metodo efficiente ed economico per riordinare i successori di un nodo non terminale.

L'idea alla base di questa euristica è che una mossa risultata la migliore in una posizione, con molta probabilità è altrettanto valida in altre posizioni dove è legale.

Ad esempio in una posizione tipica degli scacchi sono circa 35 le mosse legali, ma solamente una piccola porzione di esse merita considerazione. Una mossa che è risultata la migliore in una certa posizione è con forte probabilità la migliore anche in posizioni "simili". L'euristica in oggetto prevede che la ricerca tenga memoria di quali mosse sono risultate migliori così da esaminarle per prime in una nuova posizione (in cui sono legali) nella speranza di migliorare l'ordinamento dei nodi interni e quindi le dimensioni della ricerca.

In particolare la tabella history ricorda per una mossa m il numero di posizioni visitate in cui m è stata riconosciuta essere la migliore oppure causa di un taglio. Tale conteggio costituisce una misura della probabilità che una mossa risulti la migliore in posizioni future.

Nei giocatori artificiali di scacchi la tabella history è implementata come segue:

• una mossa è individuata dalla coppia (casa di partenza, casa di arrivo); il numero di ingressi della tabella è dunque pari alle possibili combinazioni per tale coppia moltiplicate per 2 in modo da poter distinguere le mosse del bianco da quelle del nero:

$$\text{ingressi\_tavola\_history} = 64 \cdot 64 \cdot 2 = 8196$$

• ogni volta che una mossa risulta la migliore in una ricerca a profondità d è sommato al valore corrente della mossa un punteggio dipendente da d: ha significato attribuire maggior merito ad una mossa che è risultata migliore in una ricerca a profondità maggiore rispetto ad un'altra. Valori tipici per questo incremento sono:

$$\text{incremento}(d) = 2 \cdot d \quad \text{oppure} \quad \text{incremento}(d) = 2^d$$

Risultati sperimentali hanno dimostrato che questa euristica costituisce un'ottima base di conoscenza per i meccanismi di ordinamento delle mosse: la sua efficacia è confrontabile con quella della tabella delle trasposizioni [Sch86].

1.3.4 Numeri di cospirazione: un esempio di approfondimento selettivo

Shannon classifica i metodi di ricerca degli alberi di gioco in due categorie [Sha50]. Nella prima sono riuniti i metodi di ricerca brutale (come gli algoritmi minimax ed $\alpha\beta$), nei quali sono considerate tutte le mosse possibili fino ad una fissata profondità. Il secondo approccio è quello della ricerca selettiva nel quale è usata una base di conoscenza (dipendente dall'applicazione) per selezionare un insieme di mosse "plausibili" che saranno le sole considerate. Questa metodologia non pone limiti alla profondità di ricerca (la ricerca di posizioni quiescenti costituisce un esempio di tale approccio).

La classificazione di Shannon ha suggerito lo sviluppo di un approccio intermedio chiamato approfondimento selettivo [Sch90]. L'idea è di esplorare porzioni diverse dell'albero a profondità diverse, stabilite, in generale, da un criterio dipendente dall'applicazione. Formalmente questo tipo di ricerca non è brutale perché non tutte le mosse sono esplorate fino alla stessa profondità, ma non è neppure selettiva poiché nessuna delle alternative possibili viene ignorata (seppure alcune verranno analizzate con più "interesse").

L'algoritmo dei numeri di cospirazione di McAllester [McA85] è un esempio molto attraente di approfondimento selettivo poiché il criterio di selezione dei nodi da espandere è

indipendente dall'applicazione. L'idea del metodo è di restringere l'insieme dei valori minimax plausibili per la radice dell'albero di gioco con l'intento finale di ridurre tale insieme ad una sola unità.

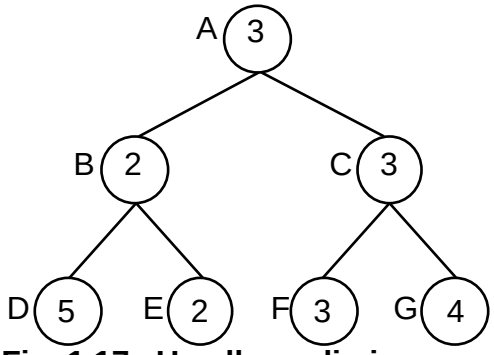


Fig. 1.17a Un albero di gioco

Valore minimax	Numero di cospirazione	Cospiratori
1	2	(D o E) e (F o G)
2	1	(F o G)
3	0	------
4	1	(E o F)
5	1	E
6	2	(D e E) o (F e G)

Fig. 1.17b Numeri di cospirazione

Un valore minimax è plausibile per la radice se il suo numero di cospirazione è inferiore ad un valore di soglia prefissato. Il numero di cospirazione per un valore minimax è definito come il minimo numero di nodi terminali dell'albero di gioco il cui valore deve cambiare (in seguito ad una ricerca più in profondità) affinché la radice assuma quel valore. Il numero di cospirazione è quindi un indice della probabilità che, esplorando più a fondo l'albero di gioco, la radice possa assumere un certo valore minimax.

Si consideri l'albero di gioco di Fig. 1.17a. Il valore minimax del nodo radice (V_{radice}) è 3. Cosa deve accadere affinché V_{radice} diventi 2?

Se il nodo F fosse esplorato più in profondità ed il suo valore minimax divenisse 2, allora anche V_{radice} avrebbe questo valore. Anche modificando il nodo G si avrebbe lo stesso effetto. È allora sufficiente che uno dei nodi terminali F o G modifichi il suo valore affinché V_{radice} divenga pari a 2. Quindi il numero di cospirazione per il valore 2 è 1, mentre F e G sono i nodi detti cospiratori. La tabella di Fig. 1.17b

elenca gli altri numeri di cospirazione per l'intervallo [1,6] di valori minimax.

Come vengono calcolati i numeri di cospirazione ?

```
#define MIN 0
#define MAX 1
int consp_n(position node,int v,int type)
{
int cn,ms,i;
m=node.minimaxvalue;
if (v==m)
       cn=0;
else if (node.depth==0)
       cn=1;
else if ((type==MAX) && (v<m) || (type==MIN) && (v>m))
       {
       cn=0;
       for(i=0;i<node.nmoves;i++)
              {
              ms=(*node.sons[i]).minimaxvalue;
              if ((type==MAX) && (ms>v) || (type==MIN) &&
(ms<v))
                     cn+=consp_n(*node.sons[i],v,!type);
              }
       }
else
       {
       cn=consp_n (*node.sons[0],v,!type);
       for(i=1;i<node.nmoves;i++)
              cn=min(cn,consp_n(*node.sons[i],!type);
       }
return(cn);
}
```

Fig. 1.18 Calcolo dei numeri di cospirazione

In un nodo di tipo MAX, come ad esempio la radice, se si vuole aumentare il valore minimax è sufficiente cambiare il valore di uno solo dei successori e quindi il numero di cospirazione associato al nuovo valore è pari al più piccolo dei numeri di cospirazione nei successori. Se invece si vuole diminuire il valore minimax portandolo a V, allora dovranno decrescere tutti i successori il cui valore è maggiore di V. Si dovrà quindi sommare i numeri di cospirazione di V in questi nodi per ottenere quello in MAX. Se il nodo è di tipo MIN vale il ragionamento inverso.

I numeri di cospirazione di un valore minimax V sono quindi calcolati ricorsivamente a partire dai nodi terminali il cui valore è 0 oppure 1 a secondo che la loro valutazione statica coincida o meno con V.

In Fig. 1.18 è illustrata questa tecnica di calcolo dei numeri di cospirazione.

Come vengono utilizzati i numeri di cospirazione?

Ad un dato istante della ricerca i valori plausibili per la radice sono ristretti all'intervallo $[V_{min},V_{max}]$. L'algoritmo sceglie, fra gli estremi dell'intervallo, quello (V) più distante dal valore attuale della radice. Successivamente l'algoritmo espande

un insieme dei cospiratori per V nel tentativo di portare a V il valore della radice. Se tale operazione ha successo deve essere ridefinito l'intervallo dei valori plausibili, altrimenti si controlla se il nuovo numero di cospirazione per V ha superato la soglia permettendo così di restringere l'intervallo di valori plausibili. Queste operazioni vengono iterate fintanto che tale intervallo non contiene un solo valore[5].

In [Sch90] è presentata una versione algoritmica completa del metodo.

Capitolo 2
GnuChess: un giocatore sequenziale di scacchi

2.1 Introduzione

GnuChess è un sistema software per la gestione di partite di scacchi in grado di giocare in modo autonomo contro un avversario umano o contro se stesso. Il sistema è strutturato in moduli programmati in linguaggio C.

GnuChess costituirà il termine di misura e confronto delle prestazioni dei giocatori paralleli che scaturiranno sia dall'applicazione dei metodi di distribuzione della ricerca che della conoscenza (Capitoli 5 e 6). Quali sono le motivazioni della scelta di questo particolare giocatore?

• GnuChess è relativamente semplice: il suo codice si compone di circa 4000 linee. Un sufficiente livello di confidenza con esso può quindi essere ottenuto in tempi piuttosto brevi (approssimativamente 15 giorni).

• le prestazioni del giocatore sono notevoli: nonostante la sua semplicità è caratterizzato da una considerevole qualità di gioco. Quanto detto trova conferma nelle vittorie da esso riportate nelle edizioni '92 e '93 del torneo annuale su piattaforma stabile fra giocatori artificiali di scacchi.

• il programma è di pubblico dominio: ciò fa di esso un terreno comune di confronto e di scambio fra molti dei ricercatori del settore.

• la struttura del programma è abbastanza modulare: questa proprietà permette il suo utilizzo come libreria di funzioni. In questa veste GnuChess (in particolare alcune delle sue funzionalità) costituirà l'interfaccia fra gli algoritmi sviluppati in questo lavoro (in generale indipendenti dal dominio di applicazione) ed il dominio in cui essi saranno sperimentati: il gioco degli scacchi.

2.2 Descrizione architetturale di GnuChess

La Fig. 2.1 descrive ad alto livello l'architettura di GnuChess utilizzando un formalismo per la rappresentazione del flusso di informazioni fra i moduli.

Modalità Di Gioco e Posizione corrente sono strutture dati globali contenenti informazioni riguardo il tipo di gioco chiesto dall'utente e lo stato corrente della partita.

Il modulo Schedulatore ordina nel tempo l'esecuzione delle due attività principali del programma: l'inserimento dei comandi o della mossa dell'utente (Interprete comandi) e la scelta della mossa migliore da parte del calcolatore (Selezione mossa). Ad esempio nella modalità di gioco: "computer vs human" le due funzionalità

sono invocate alternativamente fino a che l'utente non modifica (con alcuni comandi) la modalità di gioco corrente.

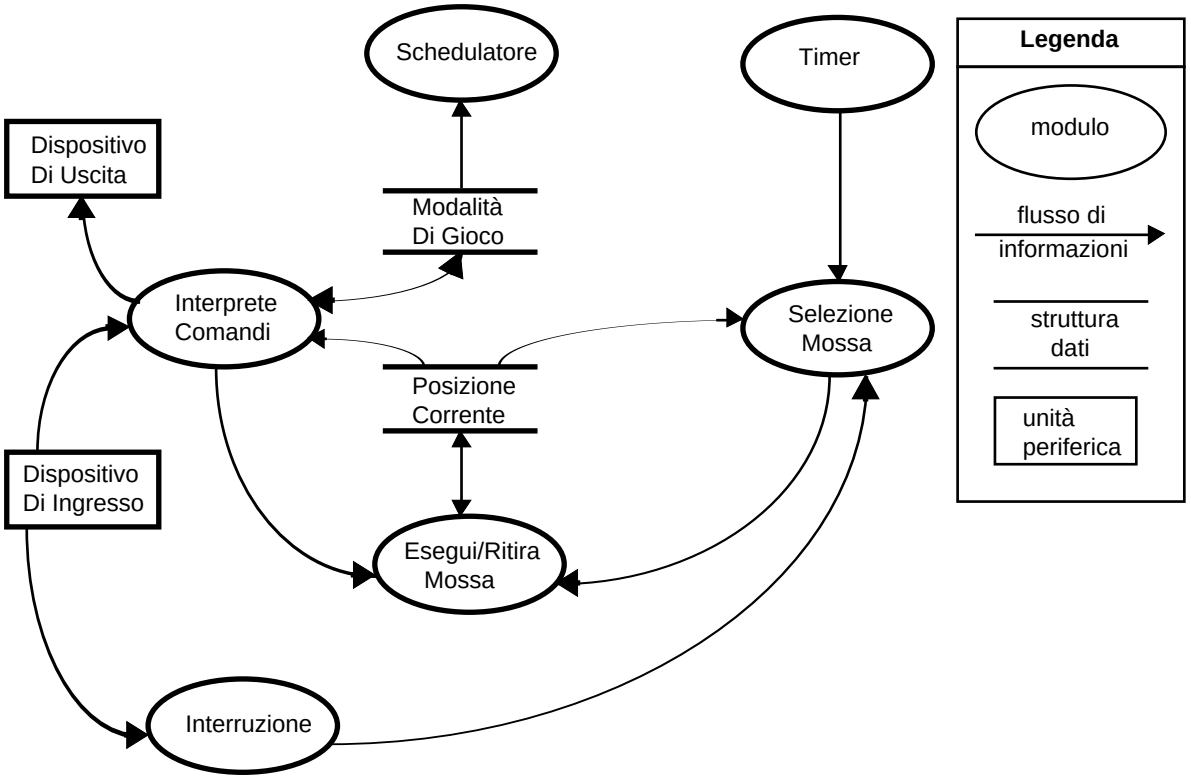


Fig. 2.1 Struttura a moduli di GnuChess

Il modulo Esegui/Ritira Mossa ha la funzione di aggiornare la posizione corrente a secondo che una mossa sia aggiunta alla lista di quelle giocate o sia invece ritirata. L'implementazione di questa funzionalità è fortemente legata alla rappresentazione della posizione.

In un generico istante solamente uno dei moduli descritti è attivo e quindi non esiste alcuna forma di parallelismo o cooperazione fra essi.

Esiste invece concorrenza fra il modulo di scelta della mossa ed i moduli gestore delle "interruzioni esterne" (Interruzione) e contatore di tempo (Timer): questi ultimi assolvono infatti la funzione di sospendere la ricerca della mossa quando l'utente lo richiede espressamente (attraverso il dispositivo d'ingresso) o è stato consumato il tempo massimo di ricerca.

I paragrafi seguenti intendono approfondire alcune scelte di progetto ed implementative fatte in GnuChess per realizzare le funzionalità più importanti di supporto ad un programma che gioca a scacchi.

2.3 La rappresentazione della posizione

È un insieme di strutture dati globali che descrivono lo stato corrente della partita.

2.3.1 La dislocazione dei pezzi in gioco

L’astrazione della scacchiera e dei pezzi disposti su essa prevede la codifica numerica delle case e dei pezzi stessi.

La generica casa è rappresentata da un valore intero contenuto nell’intervallo [0,63].

La casa identificata dall’intero sq occupa la riga R(sq) e la colonna C(sq) così definite[6]:

$R(sq) = (sq / 8) = (sq >> 3)\ R(sq) \in [0,7]$

$C(sq) = (sq \% 8) = (sq \& 7)\ C(sq) \in [0,7]$

I pezzi sono rappresentati da due informazioni aventi codifiche distinte: il tipo ed il colore. In GnuChess viene codificato esplicitamente anche il fatto che una casa sq sia vuota: essa contiene il pezzo nullo il cui colore è il neutro.

I codici di rappresentazione usati sono indicati in Tab. 2.1a e Tab. 2.1b.

La posizione viene rappresentata in modo ridondante con tecniche di associazione:

• associazione casella/pezzo: è descritta da due vettori di 64 posizioni (board e color) destinati ad associare ad ogni casa (codificata dall’indice dei vettori) rispettivamente il pezzo che la occupa ed il suo colore (rappresentati secondo il codice prima descritto).

• associazione pezzo/casella: per ogni giocatore viene gestito un vettore di riferimenti alla scacchiera (PieceList) che indica le case occupate dai suoi pezzi.

Queste informazioni, ridondanti rispetto alla associazione casella/pezzo, non sono sufficienti di per sé a rappresentare una posizione. Infatti l’indice del vettore non esprime la codifica di un pezzo, ma soltanto l’ordine con cui esso è stato individuato durante l’analisi della scacchiera.

Fa eccezione il re la cui casa è sempre contenuta nella posizione di indice 0 del vettore. L’associazione pezzo/casella è memorizzata in maniera esplicita perché molto conveniente in elaborazioni particolari come, ad esempio, la generazione delle mosse legali.

Colore	Codice
bianco	0
nero	1
neutro	2

Tab. 2.1a Codifica del colore

Pezzo	Codice
nullo	0
P	1

cavalloT	2
alfiB	3
R	4
donnaQ	5
K	6

Tab. 2.1b Codifica dei pezzi

2.3.2 Il giocatore che ha la mossa

Questa informazione non è rappresentata esplicitamente.

I giocatori sono identificati da una coppia di variabili (computer e opponent) contenenti il colore dei propri pezzi. Il giocatore cui spetta la mossa è stabilito dal flusso di programma del modulo Schedulatore. Esso infatti alterna la richiesta di esecuzione della semimossa nei confronti prima dell'uno e poi dell'altro giocatore.

2.3.3 Sequenza delle mosse giocate

Una generica semimossa viene rappresentata dalla coppia (f,t) dove f è la codifica della casa di partenza e t la codifica di quella d'arrivo del pezzo mosso. Le mosse di arrocco sono descritte (in modo non ambiguo) dallo spostamento del re.

Spesso la mossa è sintetizzata da un unico intero M ottenuto attraverso una manipolazione della codifica binaria degli interi f e t:

M = (f << 8 | t).

Questa rappresentazione assume significato se affiancata a quella della dislocazione dei pezzi: entrambe sono necessarie per identificare il pezzo mosso, l'eventuale cattura, lo scacco al re, ecc.

GnuChess tiene conto del numero di semimosse giocate (GameCnt).

In un vettore di dati strutturati (GameList) esso ricorda tutte le semimosse che hanno portato allo stato corrente del gioco.

Oltre alla coppia (f,t) per ogni semimossa sono ricordate informazioni[7] utili per la compilazione di statistiche sulla partita:

- l'eventuale pezzo catturato
- il tempo impiegato nella scelta
- se la mossa è stata scelta dal calcolatore ed in tal caso:
- la profondità della ricerca
- la valutazione numerica della mossa
- il numero di nodi dell'albero di gioco visitati.

In particolare è molto importante la conoscenza dell'ultima semimossa fatta perché consente di stabilire la legalità di

una cattura "en passant". GnuChess ricorda direttamente in una variabile globale (epsquare) la possibilità o meno di questo tipo di cattura:

epsquare=-1 ⇔ cattura en passant illegale

epsquare=sq∈ [0,63] ⇔ un pedone che può catturare nella casa sq può eseguire la cattura en passant

2.3.4 La possibilità di arrocco, la regola delle 50 mosse e la patta per ripetizione

Per ogni giocatore è memorizzato se ha eseguito la mossa di arrocco (castld). Nel caso in cui essa non sia stata giocata si deve conoscere la sua eventuale legalità.

Per questa ed altre finalità GnuChess memorizza quante volte ciascuna casa è stata occupata da un pezzo poi mosso (Mvboard). Questa informazione consente di stabilire se il re e la torre di arrocco sono mai state mosse dalla loro casa iniziale.

Una variabile (Game50) ricorda il numero ordinale dell'ultima semimossa che ha provocato una cattura, una mossa di pedone, un arrocco o una promozione. Questa informazione è indispensabile per controllare l'attivazione della regola di patta delle 50 mosse (il controllo avviene sul numero di semimosse: Gamecnt-Game50>99).

La stessa variabile è utilizzata insieme alla lista delle mosse giocate anche per stabilire l'avvenuta o meno patta per ripetizione di mosse. Infatti la variabile Game50 indica la semimossa a partire dalla quale deve partire l'analisi per il conteggio delle ripetizioni: le mosse di pedone, di cattura, di arrocco e di promozione sono infatti le sole irreversibili.

2.4 La scelta della mossa

GnuChess adotta due diversi metodi di scelta della mossa a secondo della fase della partita:

- la consultazione del libro di apertura (durante l'apertura) e
- l'analisi delle continuazioni (durante le fasi di mediogioco e di finale)

I due metodi sono mutuamente esclusivi.

2.4.1 Il libro di apertura

Durante lo stadio iniziale del gioco il programma gestisce un libro di apertura cui si fa riferimento nella scelta della mossa quando l'apertura si sviluppa lungo linee di gioco note[8].

Il libro di apertura è memorizzato in un archivio permanente dove le linee di gioco sono sequenze di mosse rappresentate in notazione algebrica. Durante il gioco una copia del libro di apertura risiede in memoria centrale.

Nel trasferimento dall'archivio alla memoria le mosse sono convertite dalla notazione algebrica per essere rappresentate nella codifica numerica interna di GnuChess descritta in 2.2.1.

Come vengono utilizzate le informazioni del libro di apertura ?

GnuChess esamina ogni linea di gioco del libro alla ricerca di una corrispondenza perfetta tra le prime mosse di questa e la sequenza di mosse che ha portato allo stato corrente del gioco. La linea di apertura per cui questo confronto ha successo suggerisce la mossa che sarà giocata dal calcolatore.

Il programma adotta un criterio di selezione casuale nel caso in cui nel libro esistano più varianti della linea di apertura seguita. Se nessuna sequenza di mosse del libro concorda con quella della partita (Book=NULL) il programma non potrà più fare uso di esso nella scelta delle mosse successive e dovrà ricorrere all'analisi delle continuazioni.

Una lacuna di GnuChess è la mancata gestione delle trasposizioni nelle linee di apertura: non sono identificate come conosciute linee di gioco che differiscono da quelle del libro per una permutazione di mosse; per poter essere considerate dovrebbero essere memorizzate esplicitamente nel libro a discapito della occupazione di memoria.

2.4.2 L'analisi delle continuazioni

Abbandonata la fase di apertura GnuChess non può più fare affidamento sui "suggerimenti" del libro di apertura, ma deve operare una valutazione degli sviluppi della partita cui possono condurre tutte le mosse possibili nella posizione corrente per poter poi scegliere la migliore di esse. Questa analisi è realizzata attraverso la visita dell'albero di gioco associato allo stato corrente.

2.4.2.1 L'albero di gioco

Fissata una posizione l'albero di gioco ne rappresenta tutte le possibili continuazioni: i nodi descrivono le posizioni raggiunte, mentre gli archi le mosse legali che portano da una posizione alla successiva (cfr. 2.2).

In GnuChess non esiste una struttura dati ad albero che riproduce esplicitamente l'albero di gioco. Ai fini della ricerca della mossa migliore, infatti, non è conveniente rappresentare l'intero albero di gioco poiché solo una porzione di esso è significativa per lo stato corrente della sua esplorazione.

A partire dalla posizione iniziale (radice) l'albero è sviluppato contemporaneamente alla sua visita: ogni volta che deve essere valutata una nuova posizione vengono generate tutte le mosse possibili a partire da questa. Così l'albero si estende in profondità; le posizioni raggiunte saranno però rappresentate soltanto al momento della loro valutazione.

Concretamente, queste informazioni sono organizzate in un vettore (Tree) di strutture partizionate al suo interno per livelli dell'albero. Ogni struttura contiene le seguenti informazioni:

• una mossa M espressa nel formato (f,t)
• insieme di dati significativi al momento della valutazione della posizione P cui ha condotto la mossa M:

• valore corrente della valutazione di P (score)
• migliore mossa di risposta dell'avversario (reply) a quella finora risultata più conveniente per il calcolatore

• insieme di indicatori (flag) di caratteristiche notevoli della mossa M e della posizione P: cattura, promozione, scacco, ecc.

Quando è completata la valutazione di una posizione, la struttura relativa scompare ed è ritirata la mossa che aveva portato ad essa. Durante la ricerca, pertanto, è rappresentata solamente la posizione esaminata ad un dato istante, mentre le altre posizioni ancora da analizzare dell'albero saranno ottenute da questa con tecniche di tipo fai/disfai (backtracking).

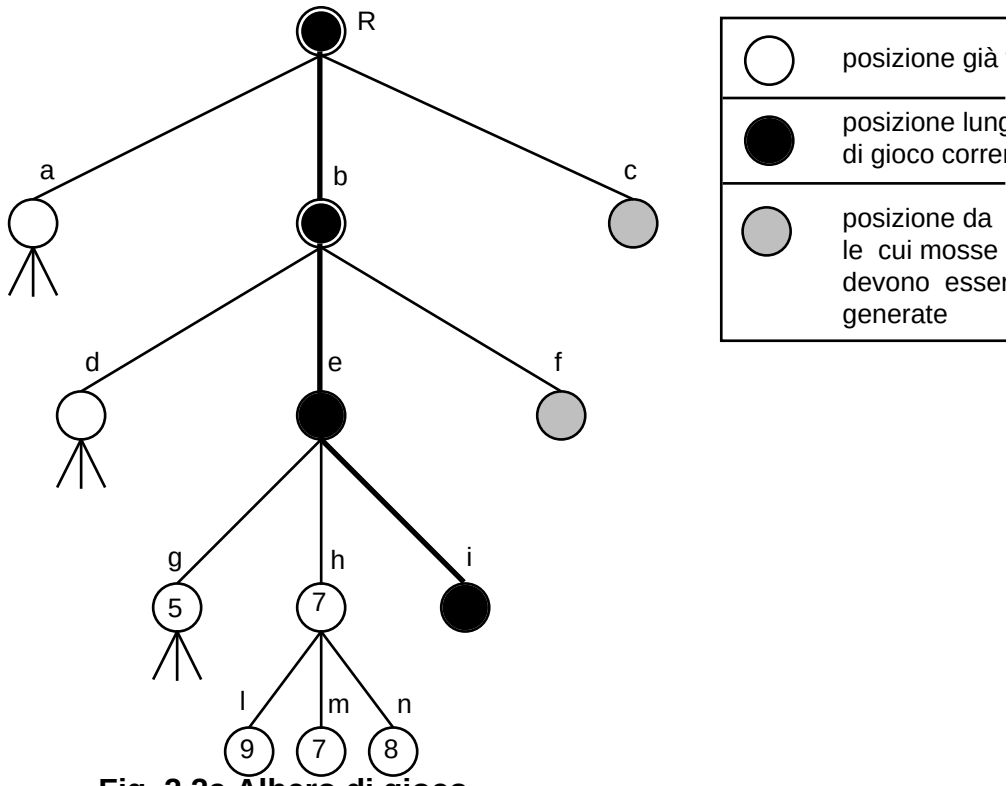


Fig. 2.2a Albero di gioco

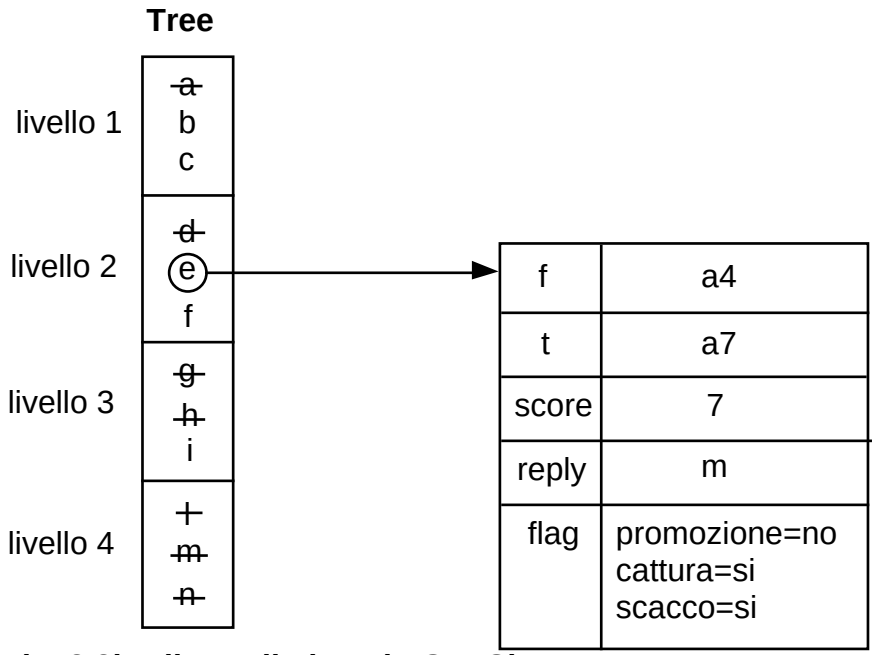


Fig. 2.2b Albero di gioco in GnuChess

Fig. 2.2b fornisce un'istantanea di queste strutture dati in una fase intermedia della visita dell'albero di gioco di Fig. 2.2a.

2.4.2.2 La generazione delle mosse

L'operazione cardine nella creazione dell'albero di gioco è la generazione di tutte le mosse possibili in una data posizione. La struttura dati fondamentale su cui opera il modulo generatore di mosse è dunque la rappresentazione della posizione.

In GnuChess la generazione riguarda un insieme di mosse pseudolegali: oltre alle mosse legalmente possibili questo insieme comprende anche quelle che lasciano il proprio re sotto scacco. Quest'ultime non vengono eliminate in fasi successive, ma concorrono alla individuazione di situazioni di matto o di stallo in cui esse sono le uniche mosse pseudolegali possibili; naturalmente alle posizioni che queste determinano vengono associati valori numerici (10001-ply, dove ply è la distanza della posizione dalla radice in termini di mosse) tali che esse non possano essere mai selezionate.

Il metodo di calcolo delle mosse implementato da GnuChess è molto efficiente ed è basato su insiemi di mappe dei movimenti [Cia92].

L'idea generale dell'algoritmo è di calcolare una grande quantità di informazioni prima che il gioco abbia inizio. I dati che vengono precalcolati sono tutte le mosse possibili per ogni pezzo a partire da una casa qualsiasi e senza tenere conto della presenza di

altri pezzi sulla scacchiera. Il calcolo di questi dati utilizza i vettori di movimento i quali per ogni pezzo definiscono la relazione matematica fissa esistente fra le coordinate di partenza e l'insieme delle coordinate delle case che esso può raggiungere con uno spostamento minimo[9] (Fig. 2.3).

Le informazioni così ottenute sono contenute in un vettore del tipo:

```
struct sqdati {
     short prossimaposizione;
     short prossima direzione;
};
struct sqdati datiposizione[8][64][64];
/* datiposizione[tipo
pezzo][origine][destinazione]*/
```

Il seguente esempio spiega il significato di queste informazioni: la prima destinazione per una torre in d4 è memorizzata in

```
datiposizione[torre][d4][d4].prossimapo
sizione
```

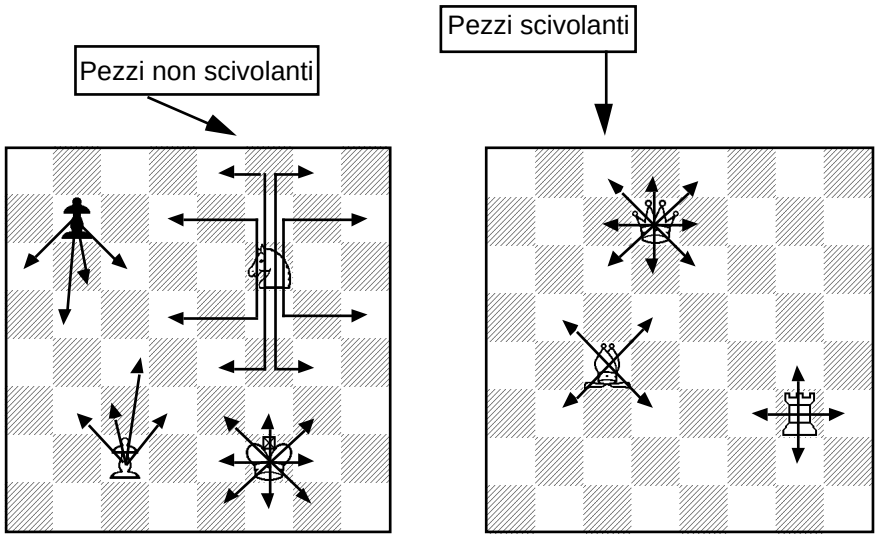
Supponiamo che questa sia d3 e che non sia occupata da un altro pezzo: la successiva casa di arrivo sarà in

```
datiposizione[torre][d4][d3].prossimapo
sizione
```

Se invece d3 è occupato da un altro pezzo la mossa seguente è in

```
datiposizione[torre][d4][d3].prossimadi
rezione
```

Grazie a questi dati l'unica cosa da fare al momento della generazione delle mosse è controllare l'eventuale collisione con altri pezzi.



Pezzo	Vettore di movimento (k è la casa di partenza)
P	k+7, k+8, k+9, k+16
T	k+6, k+10, k+15, k+17, k-6, k-10, k-15, k-17
B	k+7, k+9, k-7, k-9
R	k+1, k+8, k-1, k-8
Q	k+1, k+7, k+8, k+9, k-1, k-7, k-8, k-9
K	k+1, k+7, k+8, k+9, k-1, k-7, k-8, k-9
p	k-7, k-8, k-9, k-16

Fig. 2.3 Vettori di movimento

2.4.2.3 La scelta della migliore continuazione

L'analisi delle continuazioni del gioco a partire dalla posizione corrente prevede la visita dell'albero di gioco associato e la valutazione dei nodi incontrati su esso.

In GnuChess queste operazioni sono condotte secondo l'euristica di approfondimento iterativo (cfr. 1.3.3.1): la visita dell'albero viene ripetuta più volte, ma ad ogni iterazione la profondità del sottoalbero esplorato viene aumentata di una unità.

La prima visita è condotta fino al primo livello dell'albero; il processo iterativo termina quando è raggiunta la massima profondità di ricerca (MaxSearchDepth) oppure è scaduto il tempo massimo di ricerca (ResponseTime+ExtraTime).

Ad ogni iterazione viene prodotta una nuova valutazione numerica (score) della posizione alla radice e la continuazione principale (PrVar) che ha determinato questo valore.

GnuChess considera affidabile una valutazione che differisce di poco da quelle ottenute nelle iterazioni precedenti. Per misurare il grado di stabilità delle valutazioni ne viene calcolata una media pesata ($Zscore_{(i-1)}$) con la quale confrontare la valutazione corrente ($score_i$). Essa è definita così:

$$Zscore_i = \begin{cases} 0 & \text{se } i = 0 \\ \dfrac{Zscore_{(i-1)} + score_i}{2} & \text{se } i > 0 \end{cases}$$

dove i è il numero di iterazioni (coincide con la profondità di esplorazione dell'albero).

In GnuChess l'algoritmo di valutazione della posizione corrente scaturisce dalla combinazione del metodo

aspiration-search (cfr. 1.3.2.1) con l'algoritmo di ricerca su alberi di gioco denominato falphabeta (cfr. 1.3.2.2).

2.4.2.3.1 Aspiration-search

La tecnica aspiration-search permette di realizzare la ricerca alpha-beta con una finestra (alpha,beta) iniziale più stringente della finestra canonica (-∞,+∞) migliorando così l'efficienza della visita.

Non è tuttavia da escludere la possibilità che la ricerca non abbia successo perché il valore della radice (valore minimax) non è contenuto nella finestra iniziale; in tal caso la ricerca deve essere reiterata con una finestra che contiene sicuramente il valore minimax. Quest'ultima sarà della forma (score,+∞) se la prima ricerca ha prodotto un fallimento superiore, mentre sarà (-∞,score) se il fallimento è stato inferiore (score è il valore ritornato dalla prima ricerca).

La caratteristica fondamentale del metodo aspiration-search è la determinazione della finestra iniziale. In GnuChess essa è aggiornata ad ogni iterazione del metodo di approfondimento iterativo sulla base dei valori associati alla radice nelle iterazioni precedenti. In particolare, se alpha e beta sono gli estremi di tale finestra essi sono determinati come in Tab. 2.3:

iterazione	alpha	beta
i=1	scoreroot-90	scoreroot+90

i>1	if ($Zscore_i$<$score_i$)
	$Zscore_i$-Awndw-$zwndw_i$
	else
	$score_i$-Awndw-$zwndw_i$
		$score_i$+Bwndw

Tab. 2.3 Determinazione della finestra (alpha, beta)

Nella prima iterazione non si ha alcuna stima del valore minimax della radice e quindi il programma esegue una valutazione statica (scoreroot) della posizione corrente (è la stessa valutazione statica che sarà operata sui nodi foglia dell'albero di gioco durante la ricerca $\alpha\beta$).

Questa stima non è sicuramente affidabile perché non tiene nessun conto delle continuazioni del gioco, ma è soddisfacente se usata come centro di una finestra $\alpha\beta$.

Nelle iterazioni successive, invece, il centro della finestra è fissato dal valore minimax ricavato dall'ultima ricerca. Awndw e Bwndw sono valori costanti che definiscono l'estensione inferiore e superiore della finestra a partire dal suo centro. Si noti una certa asimmetria nella determinazione dei due estremi. GnuChess sembra infatti intenzionato a voler limitare al minimo la possibilità di un fallimento inferiore. In questo caso, infatti, il tempo massimo di ricerca viene incrementato di 10 volte il tempo inizialmente fissato; ciò non avviene nel caso di fallimento superiore. La ragione di questa scelta è che un fallimento inferiore sta ad indicare una situazione per il giocatore che ha la mossa decisamente peggiore di quanto era stato stimato e quindi deve essere prevista un'analisi molto più in profondità di tutte le possibili continuazioni. Il termine $zwndw_i$ tende appunto ad estendere inferiormente la finestra di una quantità proporzionale alla distanza rispetto al valore 0 delle valutazioni sin qui ottenute[10]:

$$zwndw_i = 20 + \left| \frac{Zscore_i}{12} \right|$$

Fig. 2.4 mostra schematicamente l'implementazione dei metodi di approfondimento iterativo e di aspiration-search in GnuChess.

```
int iterative_deepening(position *p,int
MaxSearchDepth)
{
int score,scoreroot,alpha,beta,zscore,zwndw;
scoreroot=scoreposition(p);
init_z(&zscore,&zwndw);
init_window(&alpha,&beta,scoreroot);
for (d=1;d<MaxSearchDepth && !timeout();d++)
       {
       score=falphabeta(p,alpha,beta,d);
       if (score<alpha)
              score=falphabeta(p,-
9000,score,d);
       else if (score>beta)
              score=falphabeta(p,score,9000,d);
       update_z(&zscore,&zwndw,score);
       update_window(&alpha,&beta,zscore,zwndw
,score)
       }
return (score);
```

```
}
```

Fig. 2.4 Approfondimento iterativo e aspiration-search in GnuChess

---

2.4.2.3.2 L’algoritmo di ricerca: falphabeta

Falphabeta è l’algoritmo di base usato da GnuChess per calcolare il valore minimax della radice dell’albero di gioco.

La sua particolarità è che, fissata la finestra iniziale (alpha,beta), esso determina un limite superiore più stringente (rispetto ai valori alpha o beta) per il valore corretto qualora la ricerca determini un fallimento. Questa informazione è utile perché rende più efficiente la ripetizione della ricerca stabilita da aspiration-search.

In Fig. 2.5 è descritta la versione dell’algoritmo falphabeta in GnuChess.

```
#define mate 9999
#define stalemate -9998
int falphabeta (position *p,int alpha,int
beta,int depth)

{
position *p;
int quiet,nmoves,score,best,i;
score=evaluate(p,&quiet);
if ((score==mate) || (score==stalemate))
       /* stato finale */
       return (score);
if (!quiet)   /* se la posizione è turbolenta*/
       extend_depth(&depth); /* la profondità
di ricerca aumenta */
if (depth==0)
       return (score);
nmoves=movelist(p,&successor);
best=-12000;  /* il valore minimax di un nodo è
certamente maggiore di -12000 */
if (best>alpha)
       alpha=best;
```

```
for (i=1;(i<=nmoves) &&
(best<beta);i++;successor++)
       {
       makemove(successor);
       score=-falphabeta(successor,-beta,-
alpha,depth-1)
       undomove(successor);
       if (score>best)
              {
              best=score;
              if (best>alpha)
                     alpha=best;
              }
       }
return (best);
)
```

Fig. 2.5 Algoritmo falphabeta in GnuChess

L’ordine di visita dell’albero di gioco è a scandaglio.

Un aspetto interessante di questa visita è che la funzione di valutazione è applicata anche ai nodi interni. La valutazione statica di un nodo, oltre che restituire una stima del suo valore minimax, determina informazioni di carattere generale sullo stato del gioco che saranno utilizzate da alcune delle euristiche implementate nella ricerca.

2.4.2.3.3 Euristiche per migliorare la ricerca

GnuChess completa l’algoritmo falphabeta con un certo numero di metodi finalizzati a migliorare l’affidabilità e l’efficienza della ricerca.

2.4.2.3.3.1 L’estensione della profondità di visita: la ricerca quiescente

Il programma implementa un criterio di estensione della profondità nominale di ricerca. La motivazione del metodo è di ottenere una valutazione più affidabile delle posizioni turbolente (instabili) analizzandone continuazioni più lunghe.

L’analisi di turbolenza di una posizione utilizza informazioni dedotte al momento dell’esecuzione della mossa che ha portato alla posizione (MakeMove) e durante la valutazione statica di quest’ultima. In questa analisi GnuChess distingue fra posizioni interne e terminali nell’albero di gioco.

Sia N una posizione posta al livello Ply in una ricerca di profondità nominale Sd.

Si supponga in generale che la ricerca nel sottoalbero di N sia già stata estesa e sia D (D≥Sd-Ply) la sua profondità.

Sia inoltre S la valutazione statica di N e (α, β) la finestra corrente.

N è instabile se è non terminale e:

• il re è in scacco e Ply=D-1 (si intende dare all'avversario un'altra possibilità di scacco)

• c'è una minaccia di promozione da parte dell'avversario

• vi è stata una ricattura nella mossa precedente e $\alpha \leq S \leq \beta$.

Se N è una posizione terminale (Ply≥D), essa è considerata instabile quando:

• $S \geq \alpha$ e il re è in scacco

• $S \geq \alpha$ e l'avversario minaccia una promozione

• $S \geq \alpha$, Ply=Sd+1 e almeno due pezzi del giocatore che ha la mossa sono inchiodati

• $S \leq \beta$ e l'avversario minaccia lo scacco matto.

In tutti questi casi la profondità del sottoalbero di N viene estesa di una unità divenendo D+1.

Le posizioni terminali per cui non sono verificate le precedenti condizioni di stabilità sono considerate comunque instabili. La ricerca quiescente riguarda però soltanto una porzione del loro sottoalbero: quella in cui i nodi sono generati da mosse di cattura o di promozione.

La ricerca quiescente non può comunque estendersi di più di 11 mosse oltre la profondità nominale Sd dell'albero di gioco.

2.4.2.3.3.2 La tabella delle trasposizioni

GnuChess gestisce due tabelle di trasposizioni: una residente in memoria e l'altra in un archivio permanente. La tabella in memoria ha carattere locale ad una partita ed il suo contenuto è

cancellato all'inizio di ogni partita. Le tabelle ricordano per le posizioni valutate il valore minimax o una sua limitazione (inferiore o superiore).

L'accesso alle tabelle è realizzato in modo associativo tramite una funzione hash da posizioni in interi. Tale funzione hash è così definita: prima che la partita abbia inizio viene generato un numero pseudo-casuale[11] per ogni pezzo e per ogni casa che esso può occupare.

Data la dislocazione dei pezzi e la codifica binaria dei numeri associati alla posizione di ciascun pezzo, viene calcolato lo XOR di tali valori ottenendo così un unico intero che è l'indice della tabella cercato. Con la stessa tecnica viene associato ad ogni posizione un identificatore numerico che permette di controllare il fenomeno delle collisioni nell'accesso hash.

Ogni elemento della tabella è strutturato nei seguenti campi:

• hashbd: è l'identificatore della posizione memorizzata. Se accedendo alla tabella l'identificatore della posizione corrente non coincide con il valore di questo campo si deve accedere all'elemento di indice successivo. Questa operazione può essere ripetuta un numero limitato di volte (rehash=6)

• score: è il valore calcolato per la posizione memorizzata

• flag: dice se score è il valore minimax corretto (truescore) oppure una sua limitazione inferiore (lowerbound) o superiore (upperbound)

• depth: è la profondità dell'albero di gioco esplorato nel calcolo di score

• mv : è la mossa che ha generato la posizione.

Le informazioni contenute nella tabella sono usate dall'algoritmo di ricerca solo se il contenuto del campo depth è ≥ della profondità del sottoalbero da esplorare. In questa eventualità esse permettono di restringere la finestra di

ricerca (flag==lowerbound || flag==upperbound) o di sopprimere la visita (flag==truescore).

La tabella nell’archivio permanente è acceduta nel caso in cui quella in memoria non contenga informazioni sulla posizione corrente. Tuttavia, poiché l’accesso all’archivio è molto costoso in termini di tempo, esso contiene solamente la valutazione di sottoalberi molto profondi (depth>5) e che riguardano lo stadio iniziale del gioco (GameCnt<12): le posizioni di questa fase, infatti, sono le sole ad avere una probabilità significativa di ripresentarsi in partite differenti.

2.4.2.3.3.3 L’ordinamento delle mosse

GnuChess impiega alcune euristiche per l’ordinamento delle mosse generate. L’idea è di analizzare per prime le mosse "migliori" in modo da confutare velocemente le altre alternative sfruttando la proprietà di taglio dei sottoalberi dell’algoritmo alphabeta. Il programma gestisce alcune strutture dati che rappresentano una base di conoscenza per stabilire l’efficacia di ogni mossa possibile.

Una di queste strutture dati è la variante principale: il vettore (PrVar) delle mosse che compongono la migliore continuazione trovata nella iterazione precedente del metodo di approfondimento iterativo.

GnuChess gestisce anche una lista di killer (killr1, killr2, kill3) per ogni livello dell’albero di gioco. Per killer si intende una mossa che è risultata la migliore in una posizione e che non è di cattura dell’ultimo pezzo mosso (come si vedrà queste mosse sono trattate separatamente attribuendo ad esse maggiore valore). In particolare killr1 e killr2 ricordano killer che hanno prodotto un taglio nell’albero, mentre le mosse di tipo killr3 non hanno questa proprietà.

La tabella history è costituita da un numero fisso di posizioni (8192) contenenti un intero; esse possono essere accedute in modo associativo.

Ogni elemento della tabella rappresenta una mossa codificata in termini della casa di partenza, della casa di arrivo e del colore del pezzo mosso (cfr. 1.3.3.5).

Per ogni mossa si da una misura della sua qualità in termini del numero di volte che essa è stata scelta come migliore. In particolare se J è la mossa stabilita come migliore dalla esplorazione di un albero di profondità D, history[J] viene incrementato di $2 \cdot D$ unità (si assegna maggior merito a mosse scelte da esplorazioni più profonde in quanto più affidabili).

I contenuti della continuazione principale, delle liste di killer e della tabella history sono inizializzati (=0) ogni volta che il calcolatore deve selezionare una nuova mossa.

Durante la fase di generazione ad ogni mossa viene assegnato un punteggio che permette di ordinare le mosse al momento della visita dell'albero di gioco. Tale valore viene calcolato sommando più punteggi acquisiti dalla mossa per ognuna di certe proprietà di cui può eventualmente godere.

Sia J la mossa valutata e ply il livello dell'albero in cui è giocata; Tab. 2.4 elenca queste proprietà ed il relativo peso.

Proprietà	**Punteggio**
J è nella continuazione principale	2000
promozione con donna	800
promozione con torre	600
minaccia di promozione	600
promozione con cavallo	550
promozione con alfiere	500

cattura dell'ultimo pezzo mosso	500
cattura	valore[pezzo catturato]
killr1[ply]	60
killr2[ply]	50
killr3[ply]	40
killr1[ply-2] (se ply>2)	30
"history" killer	history[J]

Tab. 2.4 Punteggi per l'ordinamento delle mosse

2.4.2.3.4 La funzione di valutazione

Data una posizione la funzione di valutazione associa ad essa un valore numerico che ne sintetizza il gradimento da parte del giocatore cui spetta la mossa.

In GnuChess il metodo di valutazione è sensibile alla fase del gioco in cui è applicato: la partita è suddivisa in stadi sulla base del numero e valore dei pezzi ancora in gioco (pedoni e re esclusi).

In tale modo viene diversificato il valore di pezzi che assumono maggiore importanza in certi momenti particolari della partita (ad esempio la torre nel finale). Allo stesso modo sono modificati i pesi attribuiti ad alcune proprietà della posizione anch'essi dipendenti dalla fase della partita.

In Tab. 2.5a e Tab. 2.5b sono presentati rispettivamente i valori dei singoli pezzi e la suddivisione della partita in stadi operata da GnuChess. Il parametro stage è il fattore di correzione dei punteggi prima menzionato: si osservi che il suo valore aumenta con il progredire del gioco.

La funzione di valutazione f di GnuChess è, da un punto di vista logico, parametrica rispetto alla posizione p valutata ed al giocatore g cui spetta la mossa in p.

Essa ha struttura polinomiale; in particolare è definita dalla relazione:

$$f(p, g) = \sum_i [t_i(p, g) - t_i(p, g')]$$

dove g' è l'avversario di g e t_i un generico termine che valuta una delle caratteristiche della posizione p dal punto di vista di uno dei giocatori.

Pezzo	Valore
P	100
cavallo T	350
B	355
R	550
donna Q	1100
K	1200

Tab. 2.5a Valore dei pezzi

M = $\sum$ del valore dei pezzi sulla scacchiera eccettuato pedoni e re	**Fase del gioco**	**Fattore di correzione dei punteggi**
M > 6600	apertura	0
$1400 \leq M \leq 6600$	mediogioco	$\frac{6600-M}{520}$
M < 1400	finale	10

Tab 2.5b Suddivisione della partita in stadi

Qual è l'identità dei termini citati?
Di seguito è presentato un loro raggruppamento logico in 7 insiemi disgiunti denominati categorie di valutazione. Tale classificazione sarà presa in considerazione nel Capitolo 4 come strumento di base per lo studio di alcune metodologie di distribuzione della conoscenza.

Le categorie di valutazione di GnuChess sono dunque le seguenti:

• il materiale: è la somma del valore dei pezzi sulla scacchiera (tali valori sono quelli indicati in Fig. 2.5a).

• il valore posizionale dei pezzi: il valore di ogni pezzo è modificato per tenere conto della casa che occupa, cioè della efficacia relativa che un pezzo ha se posizionato in zone diverse della scacchiera. Questa valutazione ha inizio prima dell'esplorazione dell'albero di gioco: è infatti analizzata la posizione corrente (alla radice

dell’albero) per compilare le mappe di controllo dello spazio.

Si tratta di strutture dati che associano ad ogni casa il valore strategico che un pezzo (di un certo tipo e di un certo colore) avrebbe se occupasse quella casa. Al momento della effettiva applicazione della funzione di valutazione queste mappe permettono di ricavare direttamente il valore posizionale di un pezzo. In Fig. 2.6 è presentata la mappa di controllo dello spazio relativa al cavallo.

7	0	4	8	10	10	8	4	0
6	4	8	16	20	20	16	8	4
5	8	16	24	28	28	24	16	8
4	10	20	28	32	32	28	20	10
3	10	20	28	32	32	28	20	10
2	8	16	24	28	28	24	16	8
1	4	8	16	20	20	16	8	4
0	0	4	8	10	10	8	4	0
	0	1	2	3	4	5	6	7

Fig. 2.6 Un esempio di mappa di controllo dello spazio

In questa categoria di valutazione è anche compresa l’attribuzione di bonus per i pezzi che assumono maggiore importanza nelle fasi avanzate della partita.

È inoltre attribuito un ulteriore merito per il possesso di entrambi gli alfieri ed analogamente per la coppia di cavalli.

• mobilità e combinazioni di attacco.
La mobilità di un giocatore è il numero di mosse legali che esso ha nella posizione corrente. GnuChess limita questa valutazione solamente agli alfieri ed alle torri.

L’analisi delle combinazioni di attacco è una componente molto "offensiva" della funzione di valutazione. In particolare sono riconosciute combinazioni di alfiere e torre per la minaccia di pezzi avversari (pin e xray). Un’ulteriore caratteristica di tale categoria di valutazione è di incoraggiare il controllo con alfiere e torre delle case adiacenti il re nemico e

l’occupazione con cavallo e regina di case a minima distanza da esso.

• la sicurezza del re: sono assegnate penalità se il re può essere minacciato, se le case adiacenti sono controllate da pezzi nemici (in particolare la regina) e se non vi sono pedoni vicino ad esso.

• la struttura pedonale: sono riconosciute particolari configurazioni deficitarie della struttura pedonale come pedoni isolati, doppiati o triplicati. È invece concesso un punteggio di merito ai pedoni passati o che comunque non hanno pedoni avversari lungo la propria colonna. I pedoni centrali sono considerati più importanti degli altri e per essi viene esaminato l’avanzamento rispetto alla casa iniziale e la mobilità.

• la protezione dei pezzi: per ciascun pezzo vengono calcolati il numero di minacce avversarie ed il numero di pezzi amici che lo proteggono. L‘eventuale penalità dipende non solo da queste quantità, ma anche dal tipo dei pezzi coinvolti: ad esempio è considerata critica la situazione di un pezzo minacciato da uno di valore inferiore rispetto ad esso e rispetto ai pezzi che lo proteggono.

Una penalità aggiuntiva è prevista se il numero complessivo di pezzi indifesi supera un valore di soglia (≥2).

• relazione pezzi-struttura pedonale: il valore posizionale di ciascun pezzo viene modificato per tenere conto dell‘influenza su esso di certe configurazioni dei pedoni.

Ad esempio la torre è valorizzata se la colonna che occupa è semiaperta o aperta; il re è invece penalizzato dalla occupazione di una colonna semiaperta, dalla assenza di propri pedoni nelle file adiacenti ed in generale dalla lontananza complessiva da tutti i pedoni.

Infine, GnuChess assegna un bonus per certe posizioni finali che sa essere vincenti. Si tratta di facili finali in cui il perdente ha solo il re. In particolare il programma ha un conoscenza dei finali di:

• pedone e re contro re (KPK) e
• cavallo, alfiere e re contro re (KBNK).

Per effetto di tale bonus il giocatore è portato a sacrificare o scambiare pezzi per raggiungere rapidamente questi finali.

2.5 La modalità di gioco

Una partita può essere giocata secondo diverse modalità che possono cambiare anche tra una mossa e l'altra a discrezione dell'utente. La modalità di gioco corrente viene memorizzata in una struttura dati globale contenente le seguenti informazioni:

- computer/opponent — è il colore dei pezzi del computer e dell'avversario umano
- bothside=true — il computer gioca per entrambi i giocatori
- force=true — l'utente inserisce le mosse per entrambi i giocatori
- TCminutes — è il livello di gioco del computer. È un valore numerico che determina il tempo di ricerca: ResponseTime=TCminutes*60
- MaxSearchDepth — è la massima profondità di ricerca. Anche questo dato influenza la qualità del gioco del computer
- easy=false(true) — il computer è abilitato (disabilitato) a "pensare" durante la scelta della mossa dell'avversario
- hash=false/true — l'utilizzo delle tabelle di trasposizioni è disabilitato (abilitato)
- Book=false/true — il computer non può (può) consultare il libro di apertura
- rcptr=false/true — l'euristica della ricattura è disabilitata (abilitata)

Una modifica della modalità di gioco può essere richiesta dall'utente con un comando inserito nel dispositivo d'ingresso ed è resa effettiva dal modulo di interpretazione dei comandi.

2.6 L'interprete dei comandi

Il modulo di interpretazione dei comandi realizza l'interfaccia di ingresso con l'utente. I comandi sono ordini o informazioni che l'utente comunica al programma in una forma dipendente dalla implementazione. Mentre il modulo di scelta della mossa è comune a tutte le versioni di GnuChess, esistono moduli di interfaccia più o meno elaborati a secondo del tipo di calcolatore ed in particolare delle sue capacità grafiche.

Il modulo di interpretazione dei comandi può essere visto anche come interfaccia di uscita in quanto l'interpretazione della maggior parte dei comandi implementa funzionalità per la comunicazione di informazioni all'utente.

I comandi che possono essere inseriti sono di diverso tipo: alcuni modificano la modalità di gioco o la posizione corrente, altri riguardano la visualizzazione di informazioni, altri ancora richiedono la lettura o la memorizzazione in archivi permanenti di informazioni sulla partita. Le tabelle Tab. 2.6 (a,b,c,d,e) elencano i comandi

principali e i relativi significati raggruppati secondo questa classificazione.

quit/exit	viene terminata l'esecuzione di GnuChess
go	l'interprete dei comandi cede il controllo allo Schedulatore
mossa del giocatore	generalmente la mossa può essere inserita in diversi formati comprensibili all'utente (varianti della notazione algebrica o il trascinamento sullo schermo del pezzo mosso). GnuChess converte la mossa indicata nella sua rappresentazione interna. Questa codifica viene quindi comunicata al modulo Esegui/Ritira Mossa per rendere consistente il nuovo stato della partita

Tab. 2.6a Controllo del programma e inserimento della mossa

bd	mostra la dislocazione dei pezzi
hint	comunica la migliore mossa di risposta trovata dal computer alla mossa da lui giocata
help	illustra sinteticamente il significato dei comandi
reverse	inverte la vista della scacchiera
post	richiede che durante la ricerca venga mostrata la continuazione principale e il punteggio relativo

Tab. 2.6b La visualizzazione delle informazioni

new	inizia una nuova partita: la posizione corrente è quella iniziale
set/edit/setup	permettono di impostare una dislocazione qualsiasi dei pezzi sulla scacchiera
undo remove	i due comandi rispettivamente ritirano l'ultima semimossa e l'ultima mossa giocate

Tab. 2.6c La modifica della posizione corrente

save	permette di archiviare la posizione corrente e la storia della partita in un file il cui nome è indicato dall'utente.

get	permette di recuperare le informazioni su una partita memorizzate su archivio permanente. Viene reso corrente lo stato del gioco al momento dell'archiviazione in modo da poter proseguire la partita. Oltre a questo sono accedute altre informazioni statistiche come la lista delle mosse giocate, il tempo di ricerca e i nodi visitati dal computer nella scelta delle sue mosse.

Tab. 2.6d Gestione degli archivi delle partite

both	abilita/disabilita il computer a giocare per entrambi i giocatori
force	abilita/disabilita l'utente a giocare per entrambi i giocatori
switch	inverte di posto i due giocatori facendo sì che il computer inizi la ricerca della mossa successiva
depth level/clock	si richiede di modificare rispettivamente la profondità massima di ricerca e il tempo massimo di ricerca del computer
hash	abilita/disabilita l'utilizzo delle tabelle di trasposizioni
book	inibisce la consultazione del libro di apertura
easy	abilita/disabilita il computer a "pensare" durante la scelta dell'avversario. Quando questa modalità di gioco è attivata il computer assume che l'avversario giocherà quella che lui considera la migliore mossa di risposta (PrVar[2]) alla mossa da lui giocata. Il computer impiega il tempo di scelta dell'avversario per cercare la sua mossa successiva. L'avversario segnala che ha fatto la sua scelta inviando un segnale di tipo interruzione che arresta questa ricerca. Se egli giocherà veramente la mossa pronosticata il programma avrà già esplorato buona parte dell'albero di gioco, altrimenti tutto il lavoro andrà perduto.

Tab. 2.6e La modifica della modalità di gioco

2.6 Il ruolo di GnuChess nella tesi

Nel seguito della tesi GnuChess costituirà un utile strumento ai fini implementativi e sperimentali delle idee che saranno presentate ed approfondite nei Capitoli 5 e 6. In particolare, nell'ambito dell'approfondimento degli algoritmi paralleli di ricerca operato nel Capitolo 5, esso sarà utilizzato alla stregua di una libreria di funzioni. Gli algoritmi che saranno presentati in quella sede, infatti, pur essendo indipendenti dal dominio di applicazione, fanno tuttavia riferimento a funzionalità classiche di gestione dell'albero di gioco (generazione di mosse, esecuzione e retrazione di una mossa, valutazione statica dei nodi terminali, ecc.) le quali sono inerentemente dipendenti dal dominio. Poiché il progetto e la realizzazione di queste funzionalità esula dagli obbiettivi di questo

lavoro, esse saranno prelevate integralmente dal codice di GnuChess. Quest'ultimo, tuttavia, non è una libreria di funzioni, ma un programma di gioco finalizzato alla minimizzazione di ridondanze e costi computazionali in nome di una massimizzazione delle prestazioni: in virtù di queste esigenze la sua struttura non è perfettamente modulare e quindi ci si dovrà attendere qualche difficoltà nella "estrazione" delle funzionalità di nostro interesse prima citate; tali eventuali impedimenti saranno eventualmente denunciati nel Capitolo 7 in sede di commento conclusivo.

Nel Capitolo 6, invece, GnuChess rivestirà un duplice ruolo.
In particolare esso sarà impiegato nello sviluppo di un giocatore parallelo costituito dalla replica di sue istanze. L'unica differenza fra tali istanze di GnuChess sarà la conoscenza terminale in esse incorporata e quindi la rispettiva funzione di valutazione. Solamente questa sezione del suo programma sarà dunque modificata (tra l'altro in minima parte), mentre il resto della sua struttura rimarrà inalterato.

Le prestazioni del giocatore parallelo così ottenuto saranno valutate sulla base della sua qualità di gioco: GnuChess costituirà, quale avversario, un valido banco di prova di tali performance.

Capitolo 3
Linda

3.1 Introduzione

Linda è un linguaggio di coordinazione caratterizzato da poche semplici operazioni basate sul paradigma di programmazione parallela chiamato comunicazione generativa. Linda non è un linguaggio di programmazione completo [CarGel90]: esso va piuttosto interpretato come un insieme di oggetti ed operazioni (su essi definite) inteso per essere integrato con un linguaggio di programmazione preesistente (linguaggio di base). Il risultato di tale estensione è dunque un "dialetto" del linguaggio di base arricchito con costrutti per descrivere e controllare la concorrenza. La classificazione più appropriata per Linda è di linguaggio di coordinamento, cioè finalizzato alla gestione ed alla coesione in unico programma di più attività separate.

È più corretto pensare a Linda come ad un modello di programmazione parallela piuttosto che ad un reale supporto alla concorrenza: Linda può essere istanziato (implementato) in molteplici modi e contesti dipendenti, ad esempio, dall'architettura del sistema (uniprocessore, multicalcolatore, rete locale di calcolatori, ecc.) o dal linguaggio di base. Particolare attenzione sarà rivolta nel seguito all'analisi di una di tali istanze del modello Linda: il sistema Network C-Linda per reti di workstation [Lin90]; questo è il supporto reale utilizzato per lo sviluppo e la sperimentazione delle idee e degli algoritmi oggetto del presente lavoro.

Ciò che distingue un linguaggio di coordinazione da uno sequenziale è la disponibilità di costrutti per la creazione ed il coordinamento (comunicazione + sincronizzazione) di più flussi di esecuzione (processi).

Linda è un modello di creazione e coordinamento di processi ortogonale al linguaggio di base in cui è incorporato: non ha interesse cosa e come computa ciascun flusso di esecuzione, bensì solo il modo con cui tali flussi (visti dunque come scatole nere) sono creati ed è permesso loro di cooperare.

Il modello su cui si fonda Linda prende il nome di comunicazione generativa. Se due processi devono comunicare, essi non scambiano messaggi (modello ad ambiente locale) o condividono una variabile (modello ad ambiente globale); al contrario, il processo mittente della comunicazione genera un nuovo oggetto (chiamato tupla) che deposita in una regione accessibile ed esterna a tutti i processi (chiamata spazio delle tuple). Il processo destinatario è così in grado di accedere a tale area per prelevare la tupla e quindi con essa il contenuto della comunicazione. La creazione dei processi è trattata alla stessa maniera: un processo che intende creare un secondo processo concorrente genera una

"tupla attiva" che affida allo spazio delle tuple. Una tupla attiva esegue una sequenza di computazioni indipendente dal processo che l'ha generata, per poi trasformare se stessa in una tupla ordinaria (passiva).

Un'implicazione di questo modello è che comunicazione e creazione di processi sono due aspetti della stessa operazione: per creare un processo viene generata una tupla attiva che si trasformerà in una passiva, mentre nella comunicazione viene generata direttamente una tupla passiva. In entrambi i casi si ottiene lo stesso risultato: lo spazio delle tuple viene arricchito con un nuovo oggetto che potrà essere acceduto da qualsiasi processo interessato ad esso.

Altra proprietà rimarchevole del modello è che i dati sono scambiati nella forma di oggetti persistenti (e non di messaggi transitori). Il destinatario della comunicazione, infatti, oltre a rimuovere la tupla generata dal mittente può anche accedere ad essa senza "consumarla", cioè permettendo che essa rimanga nello spazio delle tuple dove anche altri processi potranno leggerla.

Linda permette di organizzare collezioni di tuple in stutture dati distribuite, cioè strutture accessibili simultaneamente da più processi. L'unificazione prima descritta fra la creazione dei processi e dei dati significa che si può organizzare collezioni di processi in strutture dati attive [CarGel88]: ogni processo componente la struttura dati attiva è un flusso di computazione al cui termine diviene un elemento della struttura dati passiva che è ritornata come risultato della computazione complessiva.

Le implicazioni del modello di comunicazione generativa si estendono oltre la programmazione parallela: il modello intende essere applicato ad altre forme di comunicazione che non siano la sola cooperazione fra processi dello stesso programma parallelo. Lo spazio delle tuple si presenta come un flessibile meccanismo anche per 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 [CarGel89].

Lo spazio delle tuple esiste indipendentemente dalle computazioni dei programmi (visti come scatole nere), siano essi primitive di sistema o programmi-utente scritti in un linguaggio qualsivoglia. Le attuali implementazioni di Linda, tuttavia, concentrano per il momento l'attenzione sulla sola comunicazione fra processi di uno stesso programma parallelo.

Le pagine seguenti offriranno una presentazione più accurata del modello di programmazione distribuita basato sullo spazio delle tuple; seguirà la descrizione di C-Linda e una caratterizzazione dello stile di programmazione in Linda con alcuni esempi di implementazione di strutture dati distribuite e di interazione fra i processi. Ad epilogo di questa sezione saranno prodotti alcuni cenni sulle tecniche di realizzazione di Linda, con particolare

attenzione all’implementazione su architetture parallele con memoria distribuita.

3.2 Il modello Linda di programmazione distribuita

Linda si basa sulla nozione di spazio delle tuple che costituisce l’ambiente astratto attraverso il quale più flussi di esecuzione indipendenti (processi) interagiscono.

Lo spazio delle tuple è dunque una memoria condivisa; esso consiste di una collezione di elementi strutturalmente omogenei chiamati tuple acceduti in modo associativo e non attraverso indirizzamento.

In questo ambiente le tuple possono essere inserite, lette, rimosse o valutate. Linda definisce per ciascuna di queste funzioni un’operazione che la realizza. Tali operazioni hanno proprietà di atomicità: è così garantito che più processi possano operare "simultaneamente" sullo spazio delle tuple in modo consistente.

3.2.1 Gli oggetti

3.2.1.1 Le tuple

Una tupla è una sequenza ordinata di campi. Ad ogni campo è associato un tipo di dato; i tipi di dato possibili sono quelli fissati dal linguaggio di base. Ciascun campo di una tupla contiene un valore del tipo ad esso associato ed è per questa ragione chiamato campo attuale[12].

Il valore di un campo attuale può originare da fonti diverse: il contenuto di una variabile, il risultato di un’invocazione di funzione o una costante. Una tupla è quindi il risultato di un certo insieme di computazioni finalizzate al calcolo dei valori dei propri campi.

Quando è calcolata una tupla? e da chi?
Una risposta completa a questi quesiti sarà formulata con la presentazione degli operatori per la generazione di tuple. Per il momento è importante trarre beneficio delle precedenti considerazioni per introdurre una classificazione delle tuple in due categorie:

- tuple-dato (passive)
- tuple-processo (attive).

Informalmente si può definire una tupla passiva come una tupla completamente calcolata, i cui campi contengono effettivamente un valore attuale del tipo associato al campo. Una tupla attiva, invece, è caratterizzata dall’avere un sottoinsieme dei suoi campi con un valore non definitivamente calcolato e la cui valutazione è ancora in progresso. Più chiaramente, alcuni dei campi di una tupla attiva non contengono un dato, ma del codice la cui esecuzione avrà inizio al momento della generazione della tupla.

Formalmente, le linee di codice in un campo di una tupla attiva definiscono una funzione il cui codominio è il tipo di dato associato al campo; il valore ritornato da detta funzione andrà a sostituire lo stesso codice che l'ha descritta nel campo corrispondente della tupla. Quando tutti i campi di una tupla attiva sono stati calcolati essa diviene a tutti gli effetti una tupla-dato passiva, indistinguibile dalle altre tuple-dato.

Esempio 3.1
In un programma parallelo di scacchi si vuole condividere al livello dei processi una struttura dati chiamata mappa di controllo dello spazio. Essa associa ad ogni casa della scacchiera il valore strategico che avrebbe un certo pezzo se questo la occupasse; il contenuto della mappa dipende dallo stato corrente della partita ed è quindi aggiornato periodicamente dopo un certo numero di mosse (in 3.3.2.3.4 sarà discusso un esempio di utilizzo di tale struttura dati).

In Linda è possibile implementare tale condivisione memorizzando la mappa in un insieme di tuple con la seguente struttura logica:

```
(tipo di pezzo,casa,valore strategico)
```

Un esempio di una di queste tuple è[13]:

```
("mappa",'c',51,312)      (1)
```

Questa tupla è composta da quattro campi: un vettore di caratteri, un carattere e due interi. Si tratta di una tupla passiva poiché tutti i suoi campi contengono un valore del tipo corrispondente. Il significato di questa tupla è: il valore strategico di un cavallo ('c' ≈ cavallo) nella casa identificata con il codice numerico 51 è 312.

Siano le dichiarazioni:

```
struct scacchiera {
... };
char pezzo;
int casa;
int valore(char p,int c,struct
scacchiera s);
```

La tupla (1) potrebbe ad esempio originare dalla valutazione della tupla non valutata:

```
("mappa",pezzo,casa,valore(pezzo,casa,s
))    (2)
```

Mentre il primo campo contiene un valore costante, il secondo e terzo riferiscono due variabili e l'ultimo l'invocazione di una funzione. Se la tupla (2) è inserita nello spazio delle tuple prima di essere valutata,

allora essa è un esempio di tupla attiva. In questo caso, dal momento della sua generazione, la tupla è sottoposta ad un processo di autovalutazione che la porterà a trasformarsi nella tupla passiva (1).

3.2.1.2 Lo spazio delle tuple

Tutte le tuple sono riunite logicamente all'interno di un unico ambiente astratto: lo spazio delle tuple. Esso può contenere un numero qualsiasi di copie della stessa tupla (non è quindi un insieme).

Lo spazio delle tuple può essere visto come una memoria. Tale memoria è unica per ogni programma Linda. Essa è globale ai processi ed è condivisa da essi, cioè qualsiasi processo può accedervi e ciò senza vincoli o privilegi rispetto agli altri (a meno che questi non siano previsti esplicitamente dal programmatore).

In Linda lo spazio delle tuple è l'unico e fondamentale tramite per la comunicazione e la sincronizzazione fra i processi. Tutte le interazioni vedono dunque coinvolti tre agenti: il processo mittente che interagisce con lo spazio delle tuple e quest'ultimo che interagisce con il processo destinatario.

Nella maggioranza dei modelli tradizionali di programmazione parallela l'interazione fra processi è implementata completamente all'interno del supporto a tempo di esecuzione del linguaggio ed è quindi trasparente al programmatore che vede quindi coinvolti nell'interazione due soli agenti: il mittente e il destinatario della comunicazione. In Linda, invece, l'intermediario della comunicazione (lo spazio delle tuple) è visibile al programmatore cui spetta "l'onere" di programmare le modalità della mediazione.

3.2.2 Gli operatori

Il modello Linda definisce quattro operatori fondamentali per la manipolazione delle tuple:

- out per la generazione di tuple passive
- in per la rimozione di tuple passive
- rd per la lettura di tuple passive
- eval per la generazione di tuple attive

In un'implementazione reale di Linda è tuttavia possibile incontrare ulteriori operatori che si aggiungono a quelli di base elencati. In C-Linda, ad esempio, sono presenti gli operatori inp e rdp che costituiscono una variante in forma di predicato rispettivamente di in e rd.

3.2.2.1 out

L'operatore out aggiunge una nuova tupla allo spazio delle tuple.

Un’invocazione di questo operatore ha la forma generale:

`out(`P_1`,`P_2`,...,`P_n`).`

I parametri P_1, P_2, . . ., P_n definiscono i campi della nuova tupla. Tale tupla è prima valutata in ogni suo campo e quindi inserita nello spazio delle tuple. L’operatore out non è bloccante: il processo invocante riprende la sua esecuzione non appena ha completato, nel suo ambiente, la valutazione della tupla.

Esempio 3.2
Data un certa disposizione sulla scacchiera, si supponga di voler creare nello spazio delle tuple la porzione della mappa di controllo dello spazio relativa ad un generico pezzo. Per far questo si può definire la funzione:

```
void calcola_mappa(char pezzo,struct
scacchiera s)
     {
     int i;
     for(i=0;i<64;i++)

     out("mappa",pezzo,i,valore(pezzo,i
,s));
     }
```

3.2.2.2 in

L’operatore in "tenta" di rimuovere una tupla dallo spazio delle tuple.

L’invocazione dell’operatore ha la forma generale:

`in(`P_1`,`P_2`,...,`P_n`)`

I parametri P_1, P_2, . . ., P_n definiscono un nuovo oggetto chiamato antitupla. Si tratta di uno "schema di tupla" attraverso il quale si seleziona un sottospazio di tuple il cui contenuto e struttura interna dei tipi coincidono con quello dello schema.

La struttura di una antitupla è quindi analoga a quella di una tupla: una sequenza ordinata di campi "tipati". In questo caso, però, un campo può essere attuale o formale. Un campo attuale contiene un valore del tipo ad esso associato (come per le tuple); un campo formale, invece, è un segnaposto: ha un tipo, ma non un valore. La presenza di un formale in una antitupla è segnalata dall'occorrenza di una variabile dello stesso tipo del campo.

Esempio 3.3
Un esempio di antitupla è:

```
("mappa",'c',casa,?val)
```

I primi tre campi di essa sono attuali, mentre il quarto è formale. Il tipo di quest'ultimo è quello con cui è stata dichiarata la variabile val.

La funzione di una antitupla è dunque quella di "indirizzare" tuple all'interno dello spazio delle tuple. Le tuple non hanno né indirizzo né nome: l'unico modo per essere riferite è quello di indicare uno schema (antitupla) che sia compatibile con la loro struttura e contenuto. La modalità di indirizzamento è dunque associativa; per molti aspetti essa può essere messa in analogia con quella che caratterizza il riferimento di un oggetto in una base di dati relazionale.

Resta ancora da risolvere un'importante questione: dato lo spazio delle tuple S ed una antitupla a, quali sono le tuple in S che a riferisce?

Sono quelle che rispettano le regole di corrispondenza con a.

Def. (regole di corrispondenza o matching):

Una tupla t corrisponde ad una antitupla a se:

• t ed a hanno lo stesso numero di campi e

• per ogni coppia (c_t,c_a) di campi corrispondenti posizionalmente di t ed a:

• c_t e c_a sono dello stesso tipo e

• c_t e c_a sono in accordo, cioè:

a) se ca è un attuale, allora il valore in ct è uguale a quello in ca; le regole di uguaglianza sono quelle definite dal linguaggio di base per oggetti dello stesso tipo;

b) se ca è un formale, invece, ct e ca sono sicuramente in accordo.

Esempio 3.4

Si consideri l'antitupla a:

```
("mappa",'c',51,?val)
```

dove val è di tipo int.

La tupla t:

```
("mappa",'c',51,312)
```

rispetta i vincoli di corrispondenza con a.

Lo stesso non vale per la tupla:

```
("mappa",'c',51,312.0)
```

poiché l'ultimo campo non è dello stesso tipo del corrispondente di a.

Sia ora l'antitupla b:

```
("mappa",?pezzo,51,?val)
```

Si osservi come b costituisca uno schema di tupla più generale rispetto ad a in quanto non fissa il valore del secondo campo. Come conseguenza, se Ta è

l’insieme delle tuple che corrispondono ad a e Tb l’analogo per l’antitupla b, allora Ta $\subseteq$ Tb.

Alla luce delle definizioni di antitupla e delle regole di corrispondenza è finalmente possibile descrivere la semantica (informale) dell’operatore in:

sia S lo spazio delle tuple, a una antitupla e T={t_1, t_2, . . ., t_n} l’insieme delle tuple in S corrispondenti ad a; l’esecuzione del comando in(a) si articola nelle seguenti fasi:

• selezione di una tupla corrispondente ad a: fra le tuple dell’insieme T ne viene scelta una secondo un criterio casuale[14] (nondeterministicamente). Se l’insieme T è vuoto, allora l’esecuzione del comando viene sospesa fino a che una tupla corrispondente ad a non viene inserita nello spazio S per essere quindi selezionata dal comando stesso.

• rimozione ed assegnamento attuali-formali: la tupla t selezionata viene rimossa nella sua interezza dallo spazio delle tuple. Selezione e rimozione di t devono costituire un’azione atomica: questa condizione impedisce la situazione inconsistente con le specifiche del modello in cui più processi accedono e rimuovono la stessa tupla. Inoltre, se nella antitupla a sono presenti dei campi formali, allora alle variabili che li denotano vengono assegnati i valori attuali contenuti nei campi corrispondenti di t.

Esempio 3.5
Nell’ormai familiare programma di scacchi si vuole implementare la creazione in parallelo della mappa di controllo dello spazio. Una possibile soluzione è affidare a processi diversi il calcolo relativo a pezzi diversi. Sia M un processo coordinatore con funzione di assegnare questi compiti ai processi serventi[15]. Sia W uno di questi; nel corpo di W sarà contenuto il codice:

```
in("calcola_mappa",?pezzo,?s);
calcola_mappa(pezzo,s);
```

in cui è stata utilizzata la funzione definita nell’Esempio 3.2.

Il processo W rimuove uno dei lavori previsti dal coordinatore, impedendo così che anche altri processi serventi lo accedano ed eseguano quindi inutilmente la sua stessa computazione.

Nel caso non vi siano occorrenze di lavori nello spazio delle tuple il comando in produce l’effetto di sincronizzare W con M, poiché lo sospende fino al verificarsi dell’evento "M ha depositato un nuovo lavoro nello spazio delle tuple".

3.2.2.3 rd

L’operatore rd permette di leggere il contenuto di una tupla.

La forma generale di una sua invocazione è analoga a quella dell’operatore in:

`rd(`P_1`,`P_2`,...,`P_n`)`

I parametri P_1, P_2, . . ., P_n definiscono un’antitupla. Gli effetti dell’operatore rd sono del tutto identici a quelli di in per quanto riguarda le fasi di selezione ed assegnamento attuali-formali. Tuttavia la tupla riferita rimane (immutata) nello spazio delle tuple, dove può così essere ancora acceduta.

Questa operazione, dunque, non modifica il contenuto dello spazio delle tuple; essa è utilizzata unicamente per i suoi effetti laterali: assegnamento dei formali e sincronizzazione.

Riguardo la sincronizzazione va sottolineato, infatti, che anche l’operatore rd prevede la sospensione del processo qualora nessuna tupla "corrisponda" all’antitupla che esso ha per argomento.

Esempio 3.6
Le informazioni contenute nella struttura dati definita nell’Esempio 3.1 possono essere lette con l’operatore rd eseguendo:

```
rd("mappa",'c',casa,?val);
```

se ad esempio si vuole conoscere il valore strategico del cavallo nella posizione della scacchiera riferita dalla variabile casa.

3.2.2.4 eval

L’operatore eval aggiunge una tupla attiva allo spazio delle tuple.

L’invocazione di questo operatore ha la forma:

`eval(`P_1`,`P_2`,...,`P_n`)`

Come per l’operatore out, i parametri P_1, P_2, . . ., P_n definiscono i campi della nuova tupla. Contrariamente ad esso, però, la tupla è valutata solo dopo che essa è stata inserita nello spazio delle tuple. Implicitamente ciò comporta la creazione di nuovi processi destinati alla valutazione in parallelo di tutti i campi della tupla. Solo quando tutti i suoi campi sono stati valutati la tupla diviene un’ordinaria tupla passiva che può essere finalmente acceduta nel modo usuale con gli operatori in o rd.

In dettaglio, l’esecuzione di un’operazione eval causa la seguente sequenza di attività:

• vengono stabiliti nell’ambiente del processo invocante i legami per i nomi citati esplicitamente

nella tupla. A questo punto il processo può continuare la sua esecuzione con il comando successivo ad eval.

• ogni campo della tupla che è argomento di eval è ora valutato indipendentemente ed in modo asincrono rispetto agli altri processi. La valutazione dei campi avviene in parallelo: per ognuno di essi viene quindi creato un nuovo processo. Essa ha luogo in un contesto in cui sono ereditati dall'ambiente del processo invocante la eval i legami per i soli nomi riferiti esplicitamente nell'invocazione del costrutto.

• quando ogni campo è stato valutato completamente la tupla passiva costituita dai valori così calcolati diviene parte dello spazio delle tuple.

Ancora un'osservazione: il codice per la valutazione di un campo di una tupla attiva può contenere qualsiasi costrutto Linda, anche un'ulteriore invocazione dell'operatore eval.

3.2.2.4.1 Il modello master-worker

La primitiva eval è lo strumento Linda che permette di creare processi. La pluralità di processi è normalmente finalizzata alla soluzione efficiente di un certo problema la quale scaturisce dalla cooperazione fra i processi. Quali criteri regolano tale cooperazione?

Esistono diversi modelli di cooperazione la cui applicabilità dipende dalle caratteristiche del linguaggio parallelo e del problema.

Un modello che può essere applicato ad una estesa classe di problemi e che può essere facilmente ed efficientemente attuato in Linda è il paradigma master-worker (o ad agenda) [Bal91].

L'idea alla base di questo approccio è il concetto di distribuzione dinamica del lavoro fra i processori, intendendo per lavoro un compito, un'elaborazione necessaria ai fini della soluzione finale del problema. L'architettura logica di comunicazione indotta da questo modello ha la seguente struttura:

• un processo master (o coordinatore) che genera i lavori

• uno o più processi worker (o serventi) indistinguibili che eseguono i lavori e in generale restituiscono il risultato di questi al master cui ne compete la gestione

Il modello sottintende la presenza a livello logico di 2 strutture dati:

• l’insieme dei lavori (agenda) e
• l’insieme dei risultati dei lavori

Questo modello può essere applicato a quei problemi che possono essere suddivisi in una lista (generalmente ordinata) di compiti da eseguire. I worker estraggono ripetutamente quest’ultimi e li eseguono fino al loro esaurimento.

Come può essere attuata in Linda la creazione di questa struttura di cooperazione?

È il processo master che, intendendo disporre di n serventi eseguirà:

```
for(i=0;i<n;i++)
     eval("worker",worker());
```

La programmazione della cooperazione fra master e worker sarà ampiamente discussa nei Capitoli 4 e 5 dove il paradigma omonimo costituirà lo strumento teorico di base per la soluzione dei problemi affrontati.

Esempio 3.7
In alternativa ad una soluzione fondata sul paradigma master-worker (Esempio 3.5), la creazione in parallelo della mappa di controllo dello spazio può essere realizzata implementando questa come una struttura dati viva, cioè che si "auto-calcola". In Linda una struttura viva è costituita da tuple attive, le quali saranno generate nell’esempio con un comando del tipo:

```
eval("mappa",pezzo,casa,valore(pezzo,casa,s));
```

Questo comando crea implicitamente quattro nuovi processi per la valutazione dei campi della tupla. Il processo che calcola valore(pezzo,casa,s) fa questo in un contesto in cui i nomi valore, pezzo, casa e s hanno lo stesso valore che avevano nell’ambiente del processo che ha invocato eval. Qualsiasi variabile libera nel corpo della funzione valore diversa da quelle elencate va considerata come non inizializzata perché non riferita esplicitamente nel comando eval.

3.3 Network C-Linda

C-Linda è un’istanza, un’implementazione reale del modello Linda. Esso risulta dalla integrazione del linguaggio sequenziale C con il linguaggio di coordinamento Linda. C-Linda è dunque un linguaggio di programmazione concorrente completo:

• Linda fornisce gli strumenti per cementare in un unico programma parallelo più computazioni indipendenti;

• C permette di programmare in sequenziale ciascun flusso separato di computazione.

3.3.1 Variazioni rispetto al modello

Il linguaggio di programmazione C-Linda implementa abbastanza fedelmente il modello Linda. In esso ritroviamo oggetti e strumenti oramai familiari: le tuple, lo spazio delle tuple e i quattro operatori di base: out, in, rd ed eval. Tuttavia incontriamo anche alcune novità rispetto alla definizione del modello formulata.

3.3.1.1 Gli operatori inp e rdp

È già stata sottolineata la proprietà dei costrutti in e rd di sospendere il processo che li esegue qualora non esista alcuna tupla corrispondente all'antitupla fornita loro in argomento. Il fatto che entrambi gli operatori destinati al recupero delle tuple presentino questa caratteristica può apparire una limitazione alla espressività del linguaggio.

Si supponga infatti di voler programmare la seguente situazione: un processo P deve eseguire il comando C_1 se nello spazio delle tuple è presente la tupla t, il comando C_2 altrimenti. Seppure non impossibile, descrivere questo scenario con i soli operatori Linda di base non è immediato ed intuitivo.

Quello di cui si avrebbe bisogno è la possibilità di testare la presenza o meno di una tupla nell'omonimo spazio. Per soddisfare questa esigenza C-Linda mette a disposizione due nuovi operatori: inp e rdp.

Si tratta di varianti in forma di predicato rispettivamente di in e rd. A differenza di quest'ultimi, però, inp e rdp non causano la sospensione del processo invocante in assenza della tupla cercata.

Analizziamo in dettaglio gli effetti prodotti dalla loro esecuzione:

• argomento dei due operatori è una antitupla. Essi cercano di localizzare nello spazio delle tuple una tupla corrispondente ad essa;

• in caso di ricerca fallimentare viene ritornato il valore 0, altrimenti viene eseguito l'assegnamento attuali-formali e ritornato il valore 1 (nel caso di inp si ha anche la rimozione della tupla trovata).

L'utilizzo di questi operatori deve essere fatto con cautela poiché la loro efficienza è fortemente dipendente dall'implementazione reale. In alcune situazioni, in cui vi siano particolari esigenze riguardo le prestazioni dei programmi, è quindi preferibile

rinunciare a queste forme di predicato e ricorrere a tecniche alternative (uso di tuple contatore o tuple semaforo) [CarGel90].

Esempio 3.8
Si vuole fondare l'implementazione di un giocatore parallelo di scacchi sul modello master-worker.

In una soluzione banale il processo master distribuisce lavoro ai worker fino a che ve ne sono non occupati e di seguito si assegna lui stesso del lavoro. Completata la sua fase di calcolo esso controlla se vi sono dei risultati in arrivo dai worker, eventualmente li gestisce e successivamente riesegue ciclicamente le operazioni descritte fino ad esaurimento dei lavori. In questo contesto emerge l'utilità dei nuovi operatori inp e rdp:

```
while (!fine_lavori())
     {
     while (!fine_lavori() &&
!rdp("worker_liberi,0))
          distribuisci_un_lavoro();
     if (!fine_lavori())
          esegui_un_lavoro();
     while (inp("risultato",?r))
          gestisci_risultato(r);
     }
```

3.3.1.2 Campi formali anonimi

C-Linda permette di definire un campo formale di una antitupla come anonimo; ciò significa che a tale campo non è associata alcuna variabile. Un campo formale anonimo esprime la volontà di non conoscere il contenuto del campo corrispondente di una certa tupla. L'esecuzione dei comandi in(a) o rd(a) in cui l'antitupla a contiene un campo formale anonimo comporta infatti l'omissione dell'assegnamento attuale-formale per quel campo. Il vantaggio che deriva da questo meccanismo è duplice:

• sensibile risparmio in tempo di esecuzione nel caso di campi di grosse dimensioni

• risparmio di spazio e linee di codice per la dichiarazione della variabile destinata alla ricezione del campo attuale della tupla che sarebbe altrimenti necessaria.

Esempio 3.9
In un programma parallelo basato sul paradigma master-worker talvolta accade che un lavoro L, decomposto in sottolavori affidati a processi worker, non sia più necessario (ad esempio per l'occorrere di un taglio in una visita alpha-beta di un albero di gioco)

rendendo così inutile l’esecuzione dei sottolavori. In questa eventualità il gestore del lavoro L deve rimuovere dalla lista dei lavori i sottolavori di L non ancora prelevati dai processi worker. In questa operazione non è interessante il contenuto delle tuple rimosse: è quindi appropriato l’utilizzo di campi formali anonimi:

```
while (inp("sottolavoro_L",?struct
descrizione_lavoro));
/* descrizione lavoro è un nome di tipo
e non una variabile */
```

3.3.2 L’ambiente di programmazione

Un linguaggio di coordinamento supporta la creazione di processi e la loro interazione. Entrambi questi servizi sono di solito resi disponibili direttamente dal sistema operativo in una qualche forma accessibile al programmatore. Si potrebbe quindi evitare l’utilizzo di un linguaggio di coordinamento e fare affidamento sul sistema operativo quale supporto alla programmazione concorrente. Questa strategia è semplice, ma sfortunatamente primitiva. A differenza di una libreria di primitive del sistema operativo, un linguaggio di coordinamento è supportato da un compilatore e da un ambiente di sviluppo e controllo dei programmi.

Il compilatore è il componente più importante del sistema C-Linda. È implementato come un pre-compilatore il quale trasforma le operazioni Linda in "normali" operazioni C ottenendo così un codice completamente in C che sarà successivamente tradotto dal compilatore standard di questo linguaggio.

Il compilatore C-Linda è ottimizzante, cioè mirato a tradurre i costrutti Linda in modo che la loro esecuzione sia efficiente. Vediamo un esempio di ottimizzazione volta a ridurre il tempo di ricerca di una tupla: tuple ed antituple sono partizionate in classi di equivalenza in base alla loro struttura (numero, tipo e valore dei campi). In base a tale partizione tuple di una classe non possono corrispondere ad antituple di un’altra; in questo modo il numero di confronti che devono essere fatti per la ricerca di una tupla sono notevolmente ridotti, dato che la ricerca è limitata alle tuple di una sola classe di equivalenza e non all’intero spazio delle tuple. La creazione della partizione appena descritta è completamente a carico del compilatore Linda. L’implementazione del modello Linda sarebbe improponibile in assenza di un compilatore ottimizzante dato il degrado delle prestazioni che ne deriverebbe [Lin90].

L’ambiente di programmazione C-Linda comprende anche un complesso ed efficace insieme di utilità per l’assistenza

del programmatore nello sviluppo e la correzione dei programmi [CarGel90]. Alcuni di questi strumenti sono molto ad alto livello (Tuplescope); essi possono fornire, ad esempio, una visualizzazione grafica degli oggetti (tuple) e degli agenti (processi) che compongono un programma Linda e descriverne dinamicamente l'evoluzione durante l'esecuzione.

3.4 Lo stile di programmazione

La maggioranza dei linguaggi paralleli non consente ai processi di condividere dati in maniera diretta (linguaggi ad ambiente locale). Nei linguaggi in cui tale condivisione è invece prevista (linguaggi ad ambiente globale) normalmente la consistenza degli oggetti comuni viene garantita mettendo a disposizione del programmatore operatori elementari del tipo lock-unlock con i quali si riesce a definire come sezione critica la porzione di codice relativa all'accesso ai suddetti oggetti.

La proprietà caratterizzante del modello Linda è la possibilità offerta ai processi di condividere direttamente dati e di accedere ad essi in modo consistente garantendo l'atomicità degli accessi al livello dei costrutti concorrenti, liberando così il programmatore dal tedioso (e spesso fonte di errori) lavoro di programmazione delle sezioni critiche.

In Linda le strutture dati condivise sono memorizzate come insiemi di tuple nello spazio delle tuple. Un numero qualsiasi di processi può manipolare tuple simultaneamente poiché gli operatori sullo spazio delle tuple sono definiti intrinsecamente come consistenti.

Nel contesto della presente discussione garantire la consistenza significa impedire che un processo legga o modifichi un oggetto mentre un altro processo sta modificando quest'ultimo. Una tupla non può essere alterata fisicamente; per modificare logicamente una tupla un processo deve rimuoverla con in e quindi reinserirne una versione modificata usando out. Come conseguenza di questo approccio l'operazione rd non può mai ritornare "un'istantanea" inconsistente del contenuto di una tupla: se infatti essa è eseguita mentre la tupla è modificata da un altro processo, quella tupla sarà fisicamente assente dallo spazio delle tuple e l'operazione rd sarà sospesa fino a che la tupla non sarà completamente modificata, cioè reinserita nello spazio delle tuple.

Verranno ora illustrati alcuni esempi di strutture dati distribuite, cioè strutture condivise dai processi e memorizzate nello spazio delle tuple.

3.4.1 Strutture dati distribuite

L'insieme delle strutture dati convenzionali può essere partizionato in tre categorie:

- strutture i cui elementi sono identici o indistinguibili
- strutture i cui elementi sono distinguibili attraverso un nome
- strutture i cui elementi sono riconosciuti dalla loro posizione

L’esempio più significativo nel mondo sequenziale di strutture appartenenti alla prima categoria sono i multi-insiemi (bag), cioè insiemi in cui uno stesso dato può essere replicato più volte. La seconda categoria comprende record, oggetti istanza di classi, memorie associative, collezione di asserzioni in stile Prolog, ecc. La terza categoria include vettori, liste, grafi, alberi e così via.

Ognuna di queste categorie convenzionali è rappresentabile in forma distribuita. Tuttavia la versione distribuita di queste strutture non gioca sempre lo stesso ruolo dell’analoga sequenziale. Infatti vi sono fattori non presenti nel mondo sequenziale che invece rivestono un ruolo fondamentale nella costruzione di strutture dati distribuite. Primo fra questi il problema della sincronizzazione che emerge dal fatto che più processi asincroni possono accedere simultaneamente ad una struttura distribuita.

3.4.1.1 Strutture con elementi identici o indistinguibili

Semaforo

La più elementare delle strutture dati distribuite è il semaforo.

In Linda un semaforo (a conteggio) è niente più che una collezione di elementi identici.

All’operazione V sul semaforo S corrisponde il codice Linda:

```
out("S");
```

mentre per eseguire P su S:

```
in("S");
```

Per inizializzare il semaforo al valore n si deve eseguire out ("S") n volte.

Bag

Bag è una struttura dati sulla quale sono definite due operazioni: "aggiungi un elemento" e "preleva un elemento". In questo caso gli elementi non necessitano di essere identici, ma sono manipolati secondo modalità che fanno di essi indistinguibili.

Non importante nella programmazione sequenziale, questa struttura dati ricopre un ruolo rilevante in ambiente parallelo. La versione più semplice del paradigma master-worker, infatti, è basato sulla condivisione di un bag di lavori (agenda). Un lavoro è aggiunto al bag "lavori" usando:

```
out("lavori",Descrizione_lavoro);
```

e prelevato con:

```
in("lavori",?Nuovo_lavoro);
```

3.4.1.2 Strutture con accesso per nome

Le applicazioni parallele spesso richiedono l’accesso ad una collezione di elementi distinguibili attraverso

un nome. Tale struttura dati ricorda i tipi record del Pascal o struct del C.

In Linda possiamo memorizzare ogni elemento in tuple della forma: (nome, valore).

In analogia ai tipi struct del C, la prima parte della tupla individua il nome del campo della struttura e la seconda il suo contenuto.

Per leggere un elemento i processi eseguono:

```
rd(nome,?valore);
```

e per modificarlo:

```
in(nome,?vecchio_valore);
out(nome,nuovo_valore);
```

3.4.1.3 Strutture con accesso per posizione

Questa categoria di strutture dati può essere a sua volta suddivisa in due sottogruppi in funzione dell'ordine di accesso agli elementi:

- strutture con ordine di accesso casuale
- strutture i cui elementi possono essere acceduti in un solo ordine.

3.4.1.3.1 Strutture ad accesso casuale: i vettori

In Linda un vettore può essere implementato con un insieme di tuple della forma:

```
(nome_vettore,indice,valore)
```

Estendendo questa definizione ai vettori multi-dimensionali, un vettore di n dimensioni è descritto da tuple della forma:

```
(nome_vettore,indice1,indice2,...,indicen,valore)
```

Volendo ad esempio leggere l'elemento nella colonna 3 e riga 5 di una matrice M sarà eseguito:

```
rd("M",3,5,?val);
```

3.4.1.3.2 Strutture ordinate

Le strutture dati "ordinate" costituiscono il secondo tipo di strutture i cui elementi sono acceduti per posizione. Nello spazio delle tuple è possibile costruire qualsiasi oggetto di questo tipo. L'idea è di collegare gli elementi di queste strutture attraverso nomi logici e non per indirizzi (come invece avviene per i vettori).

Vediamo alcuni esempi.

Liste

Sia C una cella "cons" che collega gli oggetti A e B; essa può essere rappresentata dalla tupla:

```
("C","cons",["A","B"])
```

Se A è un atomo avremo:

```
("A","atomo",valore)
```

Grafi

Un generico grafo G è definito dalla coppia (N, A), dove N è un insieme di nodi ed A l'insieme di archi che collegano i nodi in N. Una naturale rappresentazione di G nello spazio delle tuple vede associata ad ogni nodo una tupla del tipo:

(nome_nodo,[$nodo_1$,$nodo_2$,...,$nodo_n$])

dove [$nodo_1$,$nodo_2$,. . .,$nodo_n$] è il vettore contenente i nomi dei nodi con cui il nodo nome_nodo è collegato.

Nel caso in cui gli archi in A siano etichettati con un valore, allora la tupla conterrà un ulteriore campo:

(nome_nodo,[$nodo_1$,$nodo_2$,...,nodon], [val_1,val_2,...,val_n])

Si noti che l'implementazione descritta è possibile solo se è supportata la possibilità di memorizzare nelle tuple vettori di lunghezza variabile (ciò è vero per gli attuali sistemi C-Linda).

Code

Una struttura di tipo coda è una sequenza ordinata di elementi che possono essere inseriti da più processi (scrittori). L'ordine degli elementi è quello di inserzione: i primi ad essere immessi saranno i primi ad essere acceduti in lettura (principio FIFO). Ad un dato istante la sequenza può essere acceduta in due soli punti: la "coda" per inserire e la "testa" per estrarre un elemento.

Vi sono molte varianti per questa categoria di dati; le più interessanti per i nostri scopi sono chiamate code-in e code-read.

In una coda-in i processi (lettori) possono rimuovere l'elemento in testa alla sequenza. Se più processi tentano di rimuovere un elemento simultaneamente, l'accesso alla coda è serializzato in modo arbitrario a tempo di esecuzione. Un processo che tenta la rimozione da una coda vuota viene sospeso fino all'inserzione di un nuovo elemento.

Nel caso di una coda-read più processi possono leggere la sequenza di elementi simultaneamente: ogni lettore legge il primo elemento, quindi il secondo e così via (la lettura non comporta la rimozione). Nella eventualità di una sequenza vuota anche in questo caso i processi verranno sospesi .

I tipi di coda presentati sono facilmente definibili in Linda. In entrambi i casi la coda consiste di una serie di tuple numerate:

```
("coda",1,val1)
("coda",2,val2)
...
("coda",n,valn)
```

L'indice dell'elemento in coda alla sequenza viene memorizzato in una tupla che funge da puntatore di estrazione:

```
("coda","puntatore_coda",8)
```

Per "appendere" un nuovo elemento alla coda un processo esegue:

```
in("coda","puntatore_coda",?indice
);
out("coda","puntatore_coda",indice
+1);
out("coda",indice+1,nuovo_elemento
);
```

In una coda-in si ha bisogno anche di una tupla per memorizzare l'indice del valore in testa; la rimozione di un elemento è così implementata dal codice:

```
in
("coda","puntatore_testa",?indice)
;
out("coda","puntatore_testa",indic
e+1);
in("coda",indice,?elemento);
```

È interessante osservare che quando la coda è vuota i processi sospesi sono sbloccati nello stesso ordine di sospensione.

In una coda-read non si ha necessità di un puntatore condiviso all'elemento di testa. Ogni processo lettore gestisce un suo puntatore locale; per leggere ogni elemento della sequenza:

```
indice=1;
while (1)
     {
     rd("coda",indice++,?elemento)
;
     ...
     }
```

Analizziamo alcune specializzazioni di queste strutture dati.

Quando una coda-in è acceduta da un solo processo lettore può essere ancora omesso il puntatore globale di testa e tale processo può usare un indice locale di estrazione.

Analogamente, quando un solo processo può inserire elementi nella coda non è necessario mantenere nello spazio delle tuple il puntatore di inserzione poiché può essere reso locale al processo scrittore.

3.4.2 L’interazione fra processi

Vi sono due motivi per cui due o più processi possono interagire: comunicare dati o sincronizzarsi. Spesso le due attività si sovrappongono all’interno di un’unica interazione.

I meccanismi che Linda mette a disposizione per "descrivere" l’interazione fra processi sono estremamente flessibili. Gli esempi che seguono intendono dimostrare questa affermazione rivelando come in Linda sia possibile implementare con facilità schemi di comunicazione e sincronizzazione caratteristici di altri modelli di programmazione parallela. Gli stessi esempi faranno emergere, oltre alla versatilità di Linda, anche ulteriori proprietà caratterizzanti.

3.4.2.1 La comunicazione

Come punto di partenza per introdurre le tecniche di comunicazione in Linda consideriamo il meccanismo di comunicazione più intuitivo e conosciuto: lo scambio di messaggi. In questo schema i processi comunicano inviando l’un l’altro dei messaggi. Gli strumenti a disposizione dei processi sono due operazioni primitive: send (invia messaggio) e receive (ricevi messaggio). Normalmente l’identità del partner della comunicazione è contenuta esplicitamente in uno degli argomenti di queste primitive (comunicazione con nomi di modulo): come riferimento per la discussione sarà considerata questa tecnica. Se un processo mittente S ha un messaggio m per un destinatario R, allora S usa un comando del tipo send(R, m) ed R invoca receive(S, x), dove x è una variabile del tipo opportuno.

Lo scambio dei messaggi può essere di vario tipo in quanto la comunicazione gode di proprietà che possono essere istanziate in modi diversi (ad esempio sincronia e simmetria). Analizziamo qualche esempio notevole che rivela come Linda consenta di descrivere in modo naturale tali forme di comunicazione.

Esempio 3.10
Siano le seguenti dichiarazioni:

```
typedef char nome_modulo[];
typedef messaggio ...;
nome_modulo mittente;
messaggio m,m1,m2,x,risultato;
```

Comunicazione simmetrica e sincrona

In questa forma di comunicazione sono coinvolti due soli processi i quali si nominano reciprocamente e devono sincronizzarsi fino al completamento della comunicazione.

Si definisca una tupla-messaggio della forma:

```
(nome_mittente, nome_destinatario, messaggio)
```

Utilizzando i costrutti concorrenti di Linda questa forma di comunicazione sarà così implementata:

```
S ::
...
m = ...;
out("S","R",m);
in("R","S","ricevuto");
/* S si sincronizza con R in attesa che questo riceva il messaggio m */
...
R ::
...
in("S","R",?x);
out("R","S","ricevuto");
...
```

Comunicazione simmetrica con rendez-vous esteso

In questo caso la comunicazione è ancora sincrona, ma il messaggio di risposta del destinatario non è finalizzato soltanto alla sincronizzazione, ma contiene anche un'informazione di ritorno per il mittente. In Linda:

```
S ::
...
m = ...;
out("S","R",m);
in("R","S",?risultato);
/* in una situazione più generale S potrebbe ricevere il risultato anche non immediatamente dopo la sua richiesta */
...
R ::
...
in("S","R",?m);
risultato=f(m,...);
out("R","S",risultato);
...
```

Comunicazione asincrona

In questa forma di comunicazione il processo mittente, una volta inviato il messaggio, non si sospende in attesa che esso sia ricevuto. Per realizzare questo schema si ha bisogno di una

qualche memoria che contenga il messaggio non ancora ricevuto dal destinatario: in Linda questa memoria è messa a disposizione dallo spazio delle tuple.

Nel caso di comunicazione simmetrica ed asincrona avremo:

```
S ::
...
m1= ...;
out("S","R",m1);
/* S non si sincronizza con la
ricezione da parte di R dei suoi
messaggi */
...
m2 = ...;
out("S","R",m2);
...
R ::
...
in("S","R",?x);
...
in("S","R",?x);
...
```

Si osservi come il mittente S sia libero di inviare messaggi ad R anche se da quest'ultimo non ha ancora ricevuto suoi messaggi precedenti. In questa eventualità non possono essere fatte assunzioni sull'ordine con cui saranno ricevuti i messaggi. Affinché i messaggi siano ricevuti nello stesso ordine di invio essi dovranno essere organizzati in una struttura dati distribuita più complessa (ad esempio una coda-in con un solo produttore ed un solo consumatore).

Comunicazione asimmetrica

Questa forma di comunicazione prevede il coinvolgimento di un numero generico di processi mittenti e destinatari.

Si parla di asimmetria in ingresso quando un processo può ricevere un messaggio da uno fra più possibili mittenti. In questo caso l'implementazione Linda non obbliga la tupla-messaggio a contenere il nome del mittente, a meno che tale informazione sia necessaria al destinatario (di sicuro nel caso di comunicazione sincrona):

```
S ::
...
m = ...;
/* questa è una comunicazione
asimmetrica in ingresso sincrona */
```

```
out("S","R",m);
in("R","S","ricevuto");
...
```
R ::
```
...
in(?mittente,"R",?x);
out("R",mittente,"ricevuto");
/* R è in grado di conoscere da quale
dei mittenti possibili ha ricevuto il
messaggio e quindi a chi inviare la
tupla di sincronizzazione */
...
```
Un’altra forma di asimmetria è quella in uscita: un messaggio può essere raccolto da più processi destinatari. Per questo tipo di comunicazione si hanno due possibili semantiche:

• partner unico: la comunicazione è stabilita con uno solo uno fra più possibili destinatari. In questo caso il messaggio è contenuto in una tupla in cui non è specificato il nome del destinatario. Il reale destinatario sarà quel processo che per primo eseguirà il comando in di rimozione di tale tupla. In una situazione reale ci si aspetta che il mittente intenda eseguire una sequenza di comunicazioni di questo tipo, dove vi sarà alternanza più o meno casuale fra i processi destinatari nella ricezione dei vari messaggi. L’interazione attraverso una struttura del tipo coda-in è un caso particolare di questo tipo di comunicazione (in essa è fissato l’ordine, FIFO, di ricezione dei messaggi).

• diffusione (broadcasting): il messaggio è ricevuto da tutti i possibili destinatari. È già stata incontrata questa forma di comunicazione in occasione della presentazione della struttura coda-read. Va sottolineata la generalità di questo meccanismo in quanto può essere applicato anche nei casi in cui il mittente non conosca né l’identità né il numero dei possibili destinatari del suo messaggio. Il problema di questa struttura è che essa non prevede la rimozione automatica del messaggio quando esso è stato raccolto da tutti i destinatari. Qualora sia noto al mittente il numero n dei destinatari si può risolvere banalmente questo inconveniente generando una tupla-messaggio per ognuno di essi:

S ::
```
...
{
int i;
m = ...;
```

```
for (i=0;i<n;i++)
     out(m);
}
...
```

È chiaro che nel caso in cui la comunicazione sia sincrona[16], cioè il mittente M si sospende fino a che il messaggio non è stato ricevuto da tutti i destinatari, è necessaria un'ulteriore struttura distribuita per implementare la "sveglia" di M (ad esempio una tupla contatore):

```
S ::
...
{
short i;
m = ...;
out("contatore",n);
for (i=0;i<n;i++)
     out(m);
in("contatore",0);
/* S si sincronizza con la ricezione di
m da parte di tutti i destinatari */
}
...
R ::
...
{
int indice;
in(?x);
in("contatore",?indice);
out("contatore",indice-1);
/* (indice-1) destinatari non hanno
ancora ricevuto il messaggio */
}
     ...
```

Gli esempi presentati hanno dimostrato che in Linda possono essere implementate una grande varietà di forme di comunicazione. Ma qual è la forma di comunicazione più elementare e naturale in Linda?

Consideriamo i comandi di base per l'invio e la ricezione di un messaggio:

```
out(m);
```

e

```
in(?x);
```

Questi costrutti sono sufficienti a stabilire lo scambio completo di un messaggio. In questo caso è sottintesa una forma di comunicazione asincrona con asimmetria sia in ingresso che in uscita. Questa è la forma di comunicazione di base in Linda; qualsiasi altra può essere ottenuta specializzando quest'ultima.

3.4.2.1.1 Alcune proprietà del modello di comunicazione

Una caratteristica fondamentale del modello Linda di programmazione distribuita è l’ortogonalità della comunicazione. Da essa derivano un certo numero di proprietà caratterizzanti [Gel85].

La comunicazione in Linda è ortogonale nel senso che sia il mittente che il destinatario non hanno alcuna conoscenza dell’identità del partner. Generalmente questa caratteristica non è presente nei linguaggi di programmazione concorrente in virtù del fatto che il mittente nomina (esplicitamente o meno) il destinatario della comunicazione.

L’ortogonalità della comunicazione ha due importanti conseguenze: il disaccoppiamento in spazio e tempo dei processi. Una terza proprietà, a sua volta, trae origine dalle prime due e prende il nome di condivisione distribuita.

Disaccoppiamento in spazio

Questa proprietà si riferisce al fatto che una tupla contenuta nello spazio delle tuple può essere acceduta da un numero qualsiasi di processi con spazi degli indirizzi disgiunti. I linguaggi concorrenti di norma permettono in un costrutto di ricezione di "importare" dati originati in uno qualsiasi fra più spazi degli indirizzi diversi. Allo stesso modo Linda consente anche ad un mittente di inviare dei dati in uno qualsiasi fra più spazi degli indirizzi distinti.

Disaccoppiamento in tempo

Una tupla inserita nell’omonimo spazio tramite out o eval rimane in esso fino a che non è rimossa da un costrutto in corrispondente. Supponiamo che il processo mittente S termini la sua esecuzione senza che la tupla da lui generata sia stata ancora "consumata". Linda permette anche ad un processo R, la cui esecuzione ha inizio successivamente alla terminazione di S, di prelevare la tupla e quindi completare la comunicazione. In Linda è quindi possibile stabilire una comunicazione fra processi non solo con spazi disgiunti, ma anche disgiunti in tempo.

Condivisione distribuita

Linda consente ad n processi con spazi degli indirizzi disgiunti di condividere una qualche variabile v memorizzando questa nello spazio delle tuple. La definizione degli operatori Linda garantisce l'atomicità e quindi la consistenza degli accessi a v. Non è quindi necessario, come nei linguaggi ad ambiente locale, che una variabile condivisa sia implementata all'interno di un processo o modulo gestore.

3.4.2.2 La sincronizzazione

Non sempre l'interazione fra processi comporta un passaggio di informazioni (dati di ingresso, risultati intermedi o finali, ecc.). Talvolta essa può essere finalizzata alla sola sincronizzazione: un processo si pone in attesa che si verifichi un certo evento la cui attuazione dipende dal comportamento di altri processi. Nei linguaggi di programmazione concorrente di norma la sincronizzazione è implementata con gli stessi strumenti messi a disposizione per la comunicazione, in quanto da un punto di vista concettuale la sincronizzazione fra processi è una comunicazione poiché comporta implicitamente il passaggio dell'informazione: "si è verificato l'evento X". Anche in Linda ritroviamo questa filosofia: come per la comunicazione, è ancora lo spazio delle tuple l'ambiente di base per l'implementazione della sincronizzazione.

Esempio 3.11
Sia un gruppo di n processi. In ogni processo sia definito un punto di sincronizzazione, cioè un punto del loro flusso di esecuzione sul quale essi si sospendono in attesa del verificarsi di uno stesso evento E. Sia E = "tutti i processi del gruppo hanno raggiunto il rispettivo punto di sincronizzazione".

La situazione descritta si presenta frequentemente nelle applicazioni parallele; essa può essere facilmente tradotta in codice Linda definendo una nuova struttura dati distribuita che può essere chiamata "barriera di sincronizzazione". Si tratta di una tupla contatore destinata a tenere memoria del numero di processi che ancora devono raggiungere la barriera stessa.

La tupla sarà inizializzata con:

```
out("barriera",n);
```

Raggiunto il punto di sincronizzazione, ogni processo del gruppo eseguirà il codice:

```
in("barriera",?val);
out("barriera",val-1);
```

```
rd("barriera",0);
```
Si osservi come il verificarsi dell'evento E su cui ogni processo si sincronizza sia stato modellato con l'evento Linda: "introduzione di ("barriera",0) nello spazio delle tuple".

3.5 Alcuni cenni riguardo l'implementazione

Il modello Linda può essere implementato su vari tipi di architetture affrontando, a seconda di esse, difficoltà e problematiche di progetto di varia natura. Le righe che seguono intendono offrire una rapida rassegna di tali problematiche, con lo scopo primario di chiarire il peso computazionale reale dei meccanismi concorrenti di Linda.

Si consideri inizialmente un'architettura convenzionale di tipo uniprocessore sequenziale. L'implementazione di Linda su questo tipo di macchina è finalizzato al solo sviluppo dei programmi ed alla simulazione del comportamento che essi avrebbero su architetture parallele[17] [CCCG85].

Su questa architettura esistono implementazioni reali di C-Linda poste al livello di astrazione immediatamente superiore a quello dell'architettura e che quindi non poggiano su un sistema operativo. Questi sistemi sono composti di due parti: il compilatore ed il nucleo (o supporto a tempo di esecuzione del linguaggio) che implementa la multiprogrammazione. Ad eccezione di alcune primitive di basso livello del nucleo, il sistema è scritto in C ed il compilatore usa lo stesso C come linguaggio intermedio: è stata già sottolineata la caratterizzazione del compilatore Linda come pre-processore destinato a tradurre i costrutti concorrenti in codice C (intercalato da eventuali invocazioni delle funzionalità di nucleo).

Lo spazio delle tuple è implementato come una tabella hash globale ai processi. Ciascuna posizione della tabella può contenere al più una tupla. Ogni volta che è invocata un'operazione sullo spazio delle tuple viene calcolato l'indirizzo hash associato alla tupla (nel caso di out o eval) o all'antitupla (nel caso di in o rd)[18]. Mantenere la consistenza della tabella hash è facile: le funzionalità di nucleo, che implementano tutte le operazioni sullo spazio delle tuple, sono eseguite con priorità superiore a quella dei processi utente e così solo un processo alla volta può essere attivo nello spazio delle tuple.

Lo schema presentato può essere esteso facilmente ad un'architettura multiprocessore con memoria comune [CCCG85]. In questo caso il problema è evitare che lo spazio delle tuple divenga un collo di bottiglia, cioè che l'accesso ad esso sia serializzato anche quando i processi intendano accedere simultaneamente a sue porzioni indipendenti. La soluzione è partizionare (sotto la guida del compilatore) lo spazio delle tuple in sottostrutture ed associare ad ognuna di esse un meccanismo di lock: la serializzazione degli accessi sarà locale a tali sottostrutture e non riguarderà più l'intero spazio delle tuple. Il degrado delle prestazioni

risultante sarà tanto minore quanto maggiore sarà la cardinalità della partizione (emerge ancora chiara l'influenza dell'analisi del compilatore sull'efficienza dei programmi paralleli).

Si consideri ora un'architettura con nodi di elaborazione che non condividono memoria (tipo rete locale o multiprocessore con memoria distribuita). Implementare Linda su questo tipo di piattaforma significa risolvere due sottoproblemi:

• dove memorizzare le tuple?
• come realizzarne la ricerca?

Almeno 2 schemi implementativi appaiono plausibili [CCCG85; Gel85]. Seppure logicamente equivalenti, essi determinano però comportamenti differenti a tempo di esecuzione.

• out distribuita: in questa soluzione l'esecuzione di out(t) fa sì che la tupla t sia comunicata per diffusione ad ogni nodo di elaborazione; ogni nodo memorizza quindi localmente una copia completa dello spazio delle tuple.

L'esecuzione di in(a) su un generico nodo N_{in} innesca la ricerca locale di una tupla t corrispondente ad a. Se la ricerca ha successo, ciò viene notificato al nodo N_0 che ha originato la tupla t. Se N_0 risponde con un messaggio (diffuso a tutti i nodi) "ok per la cancellazione di t", la tupla è rimossa localmente da ogni nodo e "concessa" al processo che ha invocato in. Se però N_0 risponde a N_{in} con il messaggio "accesso negato", ciò significa che un processo in esecuzione su un altro nodo ha tentato di rimuovere t simultaneamente a N_{in} ed è a lui che N_0 ha concesso la rimozione. Se l'iniziale ricerca locale prodotta da in(a) non ha successo, allora l'antitupla a è memorizzata in una struttura dati locale; tutte le successive tuple in arrivo saranno confrontate con le antituple sospese: qualora sia verificata una corrispondenza verrà innescato il meccanismo già descritto di interazione con gli altri nodi. L'implementazione per rd è analoga a quella di in, fatta eccezione per lo scambio di messaggi con gli altri nodi: appena è individuata una tupla corrispondente, essa è ritornata immediatamente al processo invocante rd.

• in distribuita: questo secondo schema è duale al primo. Sono le antituple ad essere diffuse a tutti i nodi, mentre le tuple rimangono nel solo nodo di origine fino a che esse non sono richieste esplicitamente. L'esecuzione di out(t) origina la ricerca locale di un'antitupla a cui t corrisponda. Se tale ricerca ha successo, allora t è inviata al nodo che ha originato l'antitupla a; in alternativa, essa rimane nell'area locale fino all'arrivo di un'antitupla corrispondente.

Questa è anche la soluzione implementativa di Network C-Linda, cioè l'implementazione su rete utilizzata in questo lavoro. Normalmente si preferisce questo approccio per le implementazioni su rete a causa della non elevata affidabilità delle comunicazioni la quale potrebbe essere origine di notevole overhead al momento della "diffusione di tuple" in una soluzione con "out distribuita". Le implicazioni di questa scelta sono una notevole efficienza delle

applicazioni parallele dove le scritture nello spazio delle tuple sono più frequenti rispetto alle letture; purtroppo questo non è il caso delle applicazioni sviluppate nel presente lavoro: in 5.2.1.4.1 saranno indicate alcune tecniche di programmazione per limitare il costo complessivo delle letture.

Il primo schema ha il vantaggio di implementare molto efficientemente l'operazione rd, ma comporta una notevole occupazione di memoria poiché ogni nodo contiene una copia dello spazio delle tuple.

Un terzo schema cerca di combinare le qualità dei precedenti. L'idea è di distribuire a tutti i nodi sia le tuple che le antituple. Un nodo N che riceve una tupla t la memorizza localmente se:

• t ha dimensioni inferiori ad un certo valore k (e quindi non grava pesantemente sull'occupazione di memoria) oppure

• è gia stata acceduta precedentemente una tupla con la stessa struttura di t da parte di un processo in esecuzione su N.

Il nodo di origine di una tupla mantiene in ogni caso una sua copia locale, fintanto che essa è presente nello spazio delle tuple. In questo schema ogni nodo tiene memoria delle tuple di maggiore interesse, cioè che hanno maggiore probabilità di essere accedute da un processo allocato in quel nodo. Quando un nodo "scarta" una tupla t di cui invece avrà bisogno successivamente non interviene alcun errore logico: al momento dell'esecuzione dell'operazione in(a) o rd(a) corrispondente, l'antitupla a verrà "diffusa" a tutti i nodi ed in particolare al nodo che ha originato t il quale in risposta invierà nuovamente ad N la tupla richiesta.

3.6 Il ruolo di Linda nella tesi

La presenza di Linda in questo lavoro ha molteplici giustificazioni: oltre all'interesse suscitato dalla novità del suo modello di comunicazione ed alla conseguente esigenza di una sua valutazione all'interno di un dominio reale di applicazione, esso sarà investigato per le sue qualità espressive e per l'efficienza ottenibile nella soluzione di istanze della classe di problemi che ci proponiamo di studiare. Un giudizio sulle sue qualità espressive e di efficienza sarà espresso nella sezione dedicata alle conclusioni (Capitolo 7).

Nel Capitolo 4 Linda consentirà di descrivere alcuni algoritmi paralleli classici di ricerca su alberi di gioco.

Nel Capitolo 5, invece, il linguaggio sarà impiegato nel progetto e nella implementazione di alcuni algoritmi di decomposizione dell'albero di gioco: sarà questo il suo vero banco di prova. La diversa granularità del parallelismo degli algoritmi citati, infatti, consentirà di impegnare Linda in diverse classi di applicazioni parallele il cui principale requisito è l'efficienza.

Il Capitolo 6 concerne il progetto di un giocatore parallelo costituito da istanze (con ricerca indipendente) di uno stesso programma sequenziale (GnuChess). La cooperazione presente in questo giocatore sarà anch'essa attuata mediante lo spazio delle tuple: in

questo dominio Linda sarà impiegato nel coordinamento ad alto livello di "lunghi" flussi sequenziali di elaborazione.

Capitolo 4
Algoritmi di ricerca parallela di alberi di gioco

Questo capitolo intende presentare una rassegna di soluzioni classiche al problema della distribuzione della ricerca di alberi di gioco. La maggior parte delle idee e degli algoritmi che verranno descritti sarà ripresa nel Capitolo 5 per discuterne l'attuazione in Linda.

4.1 Il criterio di valutazione delle prestazioni degli algoritmi paralleli

Un'importante misura dell'efficienza di un algoritmo parallelo è lo speedup. Questo parametro è definito come il rapporto tra il tempo di esecuzione della implementazione sequenziale dell'algoritmo e quello della versione parallela [Sch89].

L'obiettivo ideale è di ottenere valori per lo speedup che aumentino linearmente con il numero di processori usati. Purtroppo questo risultato è difficile da ottenere per gli algoritmi paralleli di ricerca su alberi di gioco. Esistono infatti diversi motivi di degrado delle prestazioni di questa classe di algoritmi che vengono indicati con il termine overhead [Sch89].

Overhead in tempo (OT) esprime una misura quantitativa dell'overhead totale: è la perdita percentuale di speedup confrontata con lo speedup ideale. È espressa come:

$$OT = (\text{tempo con N CPU}) \cdot \frac{N}{\text{tempo con 1 CPU}}$$

4.2 Tipi di approccio alla parallelizzazione della ricerca

Esistono diverse fonti di parallelismo nella ricerca di alberi di gioco le quali hanno originato approcci completamente differenti al processo di parallelizzazione della ricerca sequenziale:

• parallelismo nell'esecuzione di attività che riguardano i nodi individualmente (ad es. valutazione statica dei nodi terminali o generazione delle mosse)

• ricerca con finestre parallele (parallel aspiration search)
• mapping dello spazio di ricerca nei processori
• decomposizione dell'albero di gioco

4.2.1 Parallelismo nella funzione di valutazione statica

Una considerevole porzione del tempo di ricerca è spesa nella valutazione statica dei nodi terminali. La qualità delle scelte strategiche operate durante il

gioco è fortemente legata alla affidabilità e quindi alla complessità della funzione di valutazione.

Spesso si sceglie di praticare una valutazione statica semplice e grossolana preferendo impiegare il tempo di ricerca nel condurre esplorazioni più in profondità nell'albero di gioco. Il tempo dedicato alla valutazione dei nodi terminali può tuttavia essere ridotto distribuendo tale operazione su più processori, ciascuno dedicato a valutare differenti termini della funzione di valutazione. I risultati del lavoro così partizionato saranno riuniti per originare un unico valore per la posizione esaminata. Chiaramente lo speedup massimo ottenibile è limitato dalla scomponibilità e modularità della funzione di valutazione implementata.

Deep-Thought [AnCaNo90] è un giocatore artificiale di scacchi che fa uso di hardware specifico per l'esecuzione della valutazione statica. In particolare esso consiste di due componenti denominati rispettivamente valutatore veloce e lento:

• l'hardware per la valutazione veloce aggiorna i termini della funzione di valutazione che possono essere calcolati in modo incrementale, cioè a mano a mano che vengono eseguite o retratte le mosse. Tale aggiornamento avviene in parallelo alle altre fasi di esplorazione interna dell'albero; quando sarà raggiunto un nodo terminale parte dei suoi termini saranno quindi già valutati.

• i restanti termini della funzione di valutazione sono calcolati dal valutatore lento quando la ricerca arriva ad un nodo terminale. Il loro calcolo è completato molto rapidamente poiché la risorsa di elaborazione ad esso dedicata utilizza un insieme di tabelle indirizzate via hardware. Ogni tabella contiene informazioni riguardanti proprietà salienti di una posizione aggiornate durante la ricerca (struttura pedonale, pedoni passati e disposizione delle torri)

4.2.2 Generazione in parallelo delle mosse

La generazione delle mosse è un'altra operazione molto frequente nella ricerca su un albero di gioco ed in generale incide pesantemente sul tempo globale di esecuzione.

Negli scacchi, ad esempio, la velocità di generazione delle mosse è il fattore che limita le dimensioni della ricerca poiché le regole che governano il movimento dei pezzi sono piuttosto complicate.

Un accorgimento desiderabile è dunque la generazione in parallelo di tutte le mosse relative ad un nodo.

In riferimento agli scacchi, ogni colonna della scacchiera potrebbe essere assegnata ad un processore; ognuno genererà tutte le mosse per tutti i pezzi disposti sulla propria colonna. Poiché ogni colonna può contenere più pezzi di un'altra è necessaria una forma di bilanciamento del carico, cioè un riassegnamento dei lavori per mantenere tutti i processori attivi. In questa tecnica un motivo di degrado delle prestazioni è dovuto al fatto che per molti nodi non è necessario generare tutte le mosse (a causa di tagli) e così parte del lavoro andrà perduto.

Il metodo descritto si presta per una sua implementazione in hardware, cioè che utilizzi componenti architetturali progettati ad hoc.

HITECH [Ebe87] è un esempio di giocatore reale di scacchi che utilizza hardware specifico per la generazione in parallelo delle mosse. In particolare è impiegata una matrice 8x8 di processori, uno per ogni casa della scacchiera: ciascun processore calcola in parallelo agli altri le mosse legali per il pezzo che occupa la casa cui è associato (ammesso che essa non sia vuota). Il generatore parallelo determina solamente la mossa successiva che deve essere esplorata.

Esso implementa inoltre l'ordinamento delle mosse: quella suggerita è la mossa con maggiore priorità; la priorità di una mossa è una stima del suo valore ed è attribuita autonomamente dal processore che l'ha generata. Ogni processore deposita la priorità della sua migliore mossa su un bus accedibile da tutti gli altri: quando il processore con la mossa a più alta priorità riconosce che nessun altro dispone di mossa migliore, esso presenta la sua mossa al modulo dedicato al controllo della ricerca.

4.2.3 Ricerca con finestre parallele (Aspiration search parallela)

L'algoritmo aspiration search sequenziale tenta di avere successo nella ricerca disponendo di una finestra iniziale (X,Y) più stringente della finestra infinita (-∞,+∞). Se la prima ricerca fallisce, allora essa è ripetuta con la finestra (-∞,X) o (Y,+∞) a secondo del fallimento (rispettivamente inferiore o superiore).

Avendo a disposizione 3 processori si potrebbe pensare di iniziare la ricerca sulle finestre (-∞,X), (X,Y) e (Y,+∞) simultaneamente. Solamente una di queste ricerche parallele troverà la soluzione, ma ciò avverrà in un tempo minore della ricerca sequenziale con finestra (-∞,+∞), poiché finestre più stringenti determinano un numero maggiore di tagli.

In generale, avendo N processori si possono condurre in parallelo N ricerche su finestre disgiunte.

Numerosi esperimenti hanno dimostrato che lo speedup del metodo è limitato da un fattore 5 o 6, indipendentemente dal numero di processori. La ragione di questa limitazione è che esiste una parte fissa dell'albero di gioco (albero minimale) che ogni processore deve visitare. Tuttavia, quando sono utilizzati pochi processori (N=2 o 3), lo speedup può anche essere superlineare (maggiore di N) [Bau78a].

4.2.4 Mapping dello spazio di ricerca nei processori

L'idea alla base di questo approccio è di stabilire una corrispondenza fissa fra uno stato della ricerca (nodo dell'albero) e un particolare processore; tale associazione è decisa da una funzione hash.

La ricerca diviene la percorrenza in parallelo di cammini multipli con generazione e trasmissione di nodi da processore a processore.

Il vantaggio di questa forma di parallelismo è visibile nei problemi in cui lo spazio di ricerca contiene molte replicazioni degli stessi stati: uno stato sarà sempre valutato dallo stesso processore e quindi, disponendo di una memoria delle valutazioni precedenti (ad es. tabella delle trasposizioni) la valutazione di tale stato sarà effettuata una sola volta.

Normalmente i nodi dell'albero sono generati durante la ricerca e quindi comunicati dinamicamente ai processori corrispondenti [EHMN90]. La limitazione di questo approccio è che a causa del trasferimento dello stato da processore a processore l'overhead di comunicazione può divenire insostenibile.

Al contrario, Stiller ha trasferito staticamente un intero spazio di ricerca in una Connection Machine [Sti91]. Il dominio è l'analisi di finali di scacchi; il suo algoritmo opera attraverso un'analisi retrogada: esso, cioè, esegue una ricerca esaustiva all'indietro a partire da posizioni di matto per risalire fino al finale che si intende studiare. Questo metodo ha permesso di risolvere posizioni finali il cui numero di mosse

necessarie a forzare lo scacco matto è superiore a quello di qualsiasi altra posizione finora risolta.

4.2.5 Decomposizione dell’albero di gioco

Mentre il metodo aspiration search parallelo assegna ogni processore alla ricerca dell’intero albero di gioco, il metodo di decomposizione divide l’albero in sottoalberi e stabilisce che sottoalberi diversi siano esplorati da processori in generale distinti. Tale approccio al parallelismo non prevede, in linea di principio, limitazioni per lo speedup. La prospettiva di prestazioni elevate ha fatto sì che i maggiori sforzi nella ricerca siano stati concentrati nella direzione tracciata da questa classe di algoritmi paralleli.

4.2.5.1 Formalizzazione e classificazione dei metodi di decomposizione.

Non esiste un modo unico di "decomporre" la ricerca di un albero di gioco: la molteplicità di soluzioni possibili rende necessaria una loro classificazione tale da evidenziare le proprietà di approcci alla decomposizione omogenei. Si considerino a riguardo le definizioni formulate di seguito.

Def. (decomposizione):

Sia A un albero di gioco costituito dall’insieme $N=\{N_1,...,N_n\}$ di nodi (interni e terminali); sia inoltre $P=\{P_1,...,P_m\}$ un insieme di processori.

Si definisce decomposizione di A in P una funzione $D: N \rightarrow P$ che associa ad ogni nodo di A uno dei processori in P.

Def. (processo esploratore e processo supervisore):

Sia D una decomposizione dell’albero di gioco A nell’insieme P di processori; sia N l’insieme di nodi in A con $N_i \in N$. Si definisce esploratore (o responsabile) di Ni (secondo D) il processore D(Ni). Sia inoltre padre:$N \rightarrow N$ la funzione che associa ad un nodo il proprio predecessore (essa considera il nodo radice padre di se stesso); è definito supervisore di Ni (secondo D) il processore $D(padre(N_i))$.

Qual è il significato di queste definizioni?

La definizione di decomposizione come funzione intende sottolineare che la valutazione di ogni sottoalbero dell’albero di gioco è "affidata" ad un solo processore chiamato esploratore.

Sia ad esempio (N_i,P_j) una delle relazioni indotte dalla funzione di decomposizione D; si osservi che Ni identifica completamente il sottoalbero S_i di cui è radice: il significato della relazione è dunque che il valore minimax (o una sua limitazione) del sottoalbero S_i deve essere calcolato dal processore P_j. La valutazione di S_i richiede la conoscenza dei valori minimax dei sottoalberi le cui radici sono i successori di N_i: la funzione D definisce un esploratore anche per ciascuno di questi sottoalberi. Sia N_s uno dei successori di N_i e P_s=D(N_s) il rispettivo esploratore: che legame esiste fra i processori P_i e P_s?

È stabilita una dipendenza di P_s nei confronti di P_i in quanto il valore del sottoalbero lui affidato è utilizzato solo da P_i ed è quindi a questo processore che deve rispondere del suo operato. In questo caso P_i viene definito supervisore di N_s ad evidenziare il ruolo che esso assume di gestore del risultato della valutazione di N_s.

La decomposizione di un albero di gioco induce, quindi, una relazione gerarchica fra i processori che può essere descritta da una struttura ad albero. Tale relazione varia dinamicamente (cioè durante la visita complessiva dell'albero) in funzione del rapporto di "parentela" esistente fra i sottoalberi visitati ad un dato istante dai processori. Può ad esempio accadere, infatti, che in una fase successiva della visita sia P_i a dover comunicare l'esito della sua ricerca a P_s in quanto il sottoalbero da lui visitato è contenuto in quello di cui P_s è responsabile.

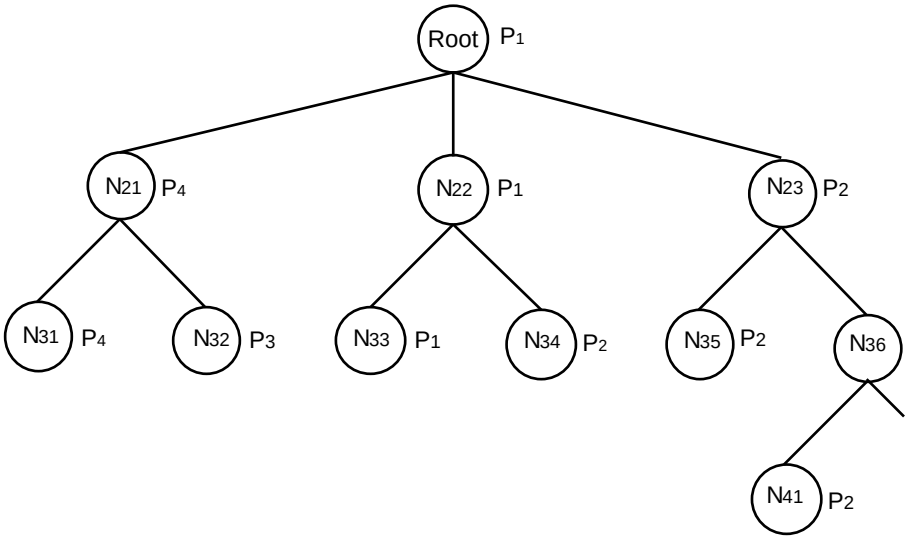


Fig. 4.19a Decomposizione: associazione nodo-processore

P_1: $N_{33} \rightarrow N_{22} \rightarrow Root$
P_2: $N_{35} \rightarrow N_{41} \rightarrow N_{23} \rightarrow N_{34}$
P_3: $N_{32} \rightarrow N_{42} \rightarrow N_{36}$
P_4: $N_{31} \rightarrow N_{21}$

Fig. 4.19b Decomposizione: ordine di valutazione dei nodi da parte dei singoli processori

---

L'albero di gioco di Fig. 4.19a descrive un esempio di decomposizione. I nodi sono etichettati con il nome del processore responsabile del sottoalbero di cui sono radice. Si osservi come alcuni processori siano responsabili di più sottoalberi. Ad un dato istante della visita, tuttavia, essi saranno impegnati nella valutazione di al più un sottoalbero. La specifica di una istanza di decomposizione deve quindi essere completata con l'ordine di valutazione, da parte di ciascun processore, dei sottoalberi di cui esso è responsabile. Tale informazione costituisce la componente "dinamica" del concetto di decomposizione; in Fig. 4.19b ne troviamo un esempio in relazione all'albero di Fig. 4.19a.

Non è in generale possibile stabilire staticamente l'ordine con cui sarà effettivamente completata la visita dei sottoalberi, in quanto legato alle velocità relative dei processori. Tuttavia si intuisce la presenza di punti di sincronizzazione che

scandiscono le fasi della visita: un processore non può completare la valutazione di un sottoalbero S fino a che non ha ricevuto la comunicazione del valore minimax di tutti i sottoalberi di S. Fig. 4.20 fornisce un'istantanea di una visita dell'albero di Fig. 4.19a consistente con la decomposizione specificata.

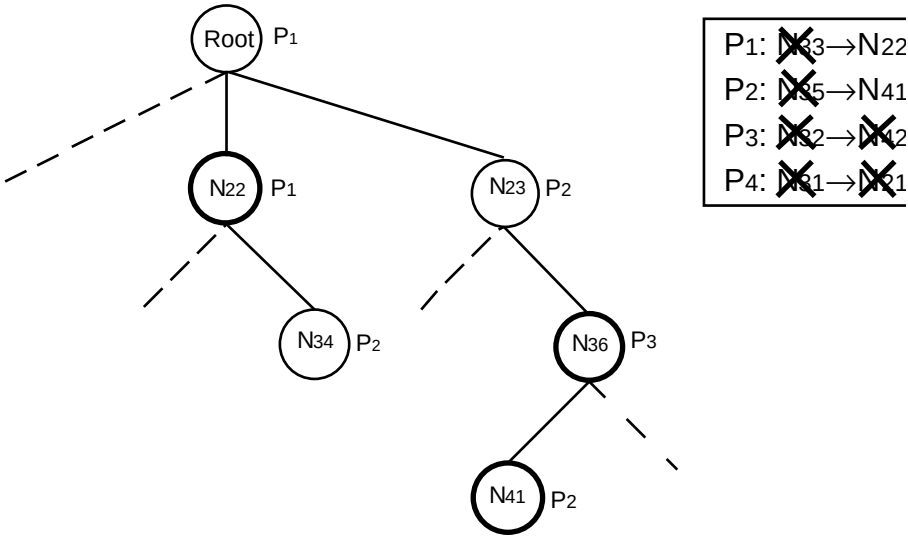


Fig. 4.20 Istantanea a tempo di esecuzione di una decomposizione

Si osservi lo stato corrente dei processori: P_4 è inoperoso in quanto ha completato la sequenza di compiti lui assegnati; P_1 è in attesa di ricevere da P_2 la valutazione del nodo N_{34}; analogamente P_3 è sospeso sulla ricezione da P_2 del valore associato a N_{41}. P_2 è dunque l'unico processore correntemente attivo: è in corso la valutazione di N_{41}. Si osservi la relazione gerarchica stabilita fra P_3 e P_2: essa sarà capovolta quando P_3, finalmente completata la valutazione di N_{36}, dovrà rispondere a P_2 del suo operato.

Dato il problema della ricerca di un albero di gioco con un metodo di decomposizione, quando è fissata l'associazione sottoalberi-processori?

In generale non ha senso che essa sia stabilita prima dell'inizio della ricerca a causa dei seguenti motivi:

• il principale problema per l'attuazione di questo approccio è che è richiesta la conoscenza della struttura dell'albero; tale

informazione è, per la maggior parte dei domini, molto costosa sia in tempo che in spazio;

• la proprietà dell'algoritmo $\alpha\beta$ di tagliare la visita di certi sottoalberi rende inutile la ricerca di alcuni di essi. Non è possibile stabilire staticamente l'identità di questi sottoalberi e quindi fissare a priori una distribuzione uniforme del carico. Lo scenario atteso al tempo della visita sarebbe il seguente: alcuni processori più "fortunati" completeranno prima degli altri la visita dei sottoalberi di cui sono responsabili (grazie alla maggiore efficacia dei tagli da essi prodotti) e resteranno inoperosi fino al completamento della visita complessiva. Qualsiasi soluzione a questo problema deve essere di natura dinamica, cioè deve prevedere meccanismi che durante la visita permettano di riassegnare a processori inoperosi sottoalberi già associati a processori sovraccarichi.

Una classificazione dei metodi di decomposizione in base al tempo in cui è fissata la decomposizione è dunque poco significativa in quanto l'unico approccio plausibile è quello di tipo dinamico. Ha invece senso distinguere le diverse soluzioni in base al tempo in cui esse stabiliscono i nodi dell'albero in cui deve avvenire una reale decomposizione.

Def. (nodo di decomposizione):
Sia D la decomposizione dell'albero A nell'insieme P di processori e Ni un nodo di A; Ni è un nodo di decomposizione (secondo D) $\Leftrightarrow \exists$Nj: Ni=padre(Nj) e D(Nj)$\neq$D(Ni).

Si definisce dunque di decomposizione un nodo in cui avviene una distribuzione della ricerca, cioè dove alcuni dei propri sottoalberi sono esplorati da processori diversi dal suo responsabile. In funzione di questa definizione può essere formulata la seguente classificazione dei metodi di decomposizione in statici e dinamici:

• un algoritmo di decomposizione statica fissa a priori l'identità di tutti i nodi di decomposizione;

• in un algoritmo di decomposizione dinamica, invece, una parte dei nodi di decomposizione è designata durante la ricerca.

Le definizioni proposte saranno recuperate nel Capitolo 5 dove costituiranno strumenti essenziali nella discussione degli algoritmi discussi in quella sede.

4.2.5.2 Motivi di degrado delle prestazioni nei metodi di decomposizione

Le principali cause della limitazione delle prestazioni nell'approccio di decomposizione sono [Sch89]:

• overhead di ricerca (OR): in ambiente sequenziale tutte le informazioni ricavate fino ad un dato punto della ricerca sono disponibili per effettuare decisioni quali il taglio di un sottoalbero.

In ambiente distribuito le stesse informazioni possono essere disperse su macchine diverse e quindi non completamente utilizzabili: ciò può portare ad una esplorazione non necessaria di una porzione dell'albero di gioco. La crescita delle dimensioni dell'albero è chiamata overhead di ricerca. Tale forma di degrado può essere approssimata osservando che la dimensione dell'albero esplorato è proporzionale al tempo di ricerca. OR è più precisamente calcolato dalla relazione:

$$OR = \frac{\text{nodi visitati da N CPUs}}{\text{nodi visitati da 1 CPU}} - 1$$

Va notato come, in talune occasioni, l'incompletezza di informazioni è vantaggiosa. Si supponga, ad esempio, che il secondo figlio, ma non il primo, di un nodo CUT determini un taglio. La versione sequenziale esplorerebbe interamente il sottoalbero più a sinistra prima di considerare il successivo e scoprire così il taglio.

Visitando i due sottoalberi in parallelo può accadere che il taglio venga individuato permettendo di arrestare rapidamente la ricerca nel sottoalbero più a sinistra. Questa osservazione potrebbe giustificare un valore dello speedup che supera il numero dei processori, ma purtroppo la situazione descritta è poco probabile poiché il taglio si

presenta più frequentemente nel primo sottoalbero.

• overhead di comunicazione (OC): è il carico addizionale che un programma parallelo deve sopportare quando è impiegato tempo non trascurabile per la comunicazione dei messaggi fra i processi. Questo costo può essere limitato in fase di programmazione scegliendo opportunamente le dimensioni e la frequenza dei messaggi. Una stima di OC è data dal prodotto fra il numero di messaggi inviati e il costo medio di un messaggio.

• overhead di sincronizzazione (OS): è il costo che occorre quando alcuni dei processori sono inattivi. In teoria tutti i processori dovrebbero essere occupati nello svolgere lavoro utile per tutto il tempo di esecuzione. Nella realtà questo comportamento ideale viene meno quando un processo deve sincronizzarsi con un altro determinando attesa attiva. Potrebbe accadere ad esempio che un processo P voglia agire su un oggetto condiviso accedibile in mutua esclusione, mentre un altro processo Q sta già operando su quell'oggetto. Il processo P rimarrà bloccato (e quindi inattivo) fintanto che Q non abbia completato la sua operazione. Tale overhead può presentarsi anche quando un processo è in attesa che certi risultati, senza i quali non può continuare, gli siano forniti da un altro.

L'overhead complessivo OT è funzione delle tre forme di overhead elencate:

OT = f (OR, OC, OS)

Per massimizzare le prestazioni di un algoritmo parallelo è necessario minimizzare i vari tipi di overhead. Purtroppo essi non sono mutuamente indipendenti e lo sforzo nella minimizzazione di uno di essi può risultare nell'aggravio di un altro. Ad esempio la riduzione dell'overhead di ricerca richiede normalmente un aumento del numero delle comunicazioni.

4.2.5.3 L'algoritmo base di decomposizione ed il concetto di gerarchia dei processori

Come introduzione ai metodi di decomposizione dell'albero di gioco si osservi l'algoritmo[19] di Fig. 4.21.

```
/* si suppone che l'albero dei processori sia
rappresentato in una forma distribuita fra i
nodi; ogni processore ha delle variabili
locali contenenti le seguenti informazioni:
ME=identificatore del processore;
NSONS=numero di sottoprocessori */
#define update_alpha(VALUE)\
{\
if (VALUE>alpha)\
       alpha=VALUE;\
if (alpha>=beta)\
       {\
       terminate(); /* implementa la
terminazione di tutti i sottoprocessori */ \
       return(alpha);\
       }\
}\
int treesplit(position p,int alpha,int
beta,int depth)
{
int nmoves,result,free,ind;
position *successor;
if (depth==0)
       return(evaluate(p));
if (NSONS==0)
       return(alphabeta(p,alpha,beta,depth));
nmoves=genmoves(p,&successor);
if (nmoves==0)
       return(evaluate(p));
for (i=0,free=NSONS;i<nmoves;ind++)
       {
       eval
("split",ME,treesplit(*(successor+ind),-beta,-
alpha,depth-1));
       free--;
       if (free==0)
              {
              in ("split",ME,?result);
              free++;
              update_alpha(-result);
              }
       }
while (free<NSONS)
       {
       in ("split",ME,?result);
       free++;
       update_alpha(-result);
       }
return(alpha);
}
```

Fig. 4.21 Algoritmo base di decomposizione

Si tratta di un algoritmo di decomposizione statica estremamente semplice il quale prevede che la distribuzione della ricerca, cioè l'attribuzione della visita di sottoalberi a processori distinti, avvenga al livello della radice.

Quella presentata è tuttavia una versione generalizzata secondo la quale un processore

può affidare a sua volta parte dello spazio di ricerca di cui è responsabile ad un suo sotto-processore: tale possibilità è dipendente dalla struttura logica e topologica dei processori utilizzati complessivamente.

In particolare ad un processore arbitrario è assegnato il compito di supervisore della ricerca. Questo distribuisce l'esplorazione del sottoalbero più a sinistra della radice al sottoprocessore P_1, il secondo sottoalbero al sottoprocessore P_2 e così via. I sottoprocessori ritornano il risultato della loro visita al supervisore, il quale sulla base di tutti i valori comunicati calcola il valore minimax dell'albero di gioco.

Ogni sottoprocessore ha a disposizione dei processori di livello ancora inferiore cui è libero di distribuire l'analisi di certe porzioni del sottoalbero assegnatogli (in questo caso il sottoprocessore si comporta da supervisore). Ovviamente ogni processore possiede un limitato numero F di sottoprocessori. Se tutti i suoi sottoprocessori sono occupati, deve attendere che uno fra questi sia nuovamente disponibile per assegnargli una parte inesplorata del suo sottoalbero.

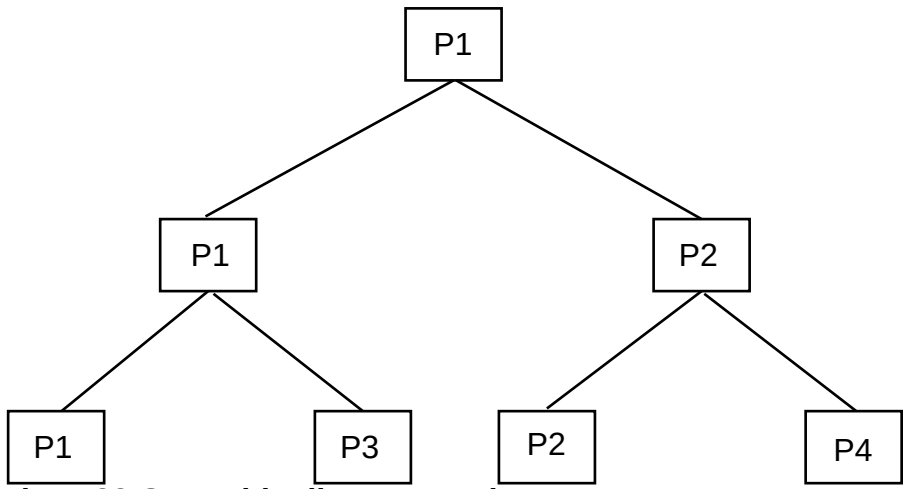


Fig. 4.22 Gerarchia di processori

L'organizzazione descritta introduce una gerarchia fra i processori descrivibile tramite un albero uniforme con fattore di diramazione F e profondità D (Fig. 4.22). Dopo D livelli di subappalto della ricerca un processore dovrà valutare da solo il sottoalbero assegnato.

Questo processore occupa un nodo foglia nell'albero dei processori ed è chiamato esploratore.

Questa organizzazione limita le comunicazioni interprocessore, ma fa diminuire la condivisione di informazioni originando così un pesante overhead di ricerca.

Un esempio di raffinamento del metodo è: poiché i processori supervisore sono lungamente inattivi in attesa che un sottoprocessore abbia completato la ricerca, parte del loro tempo di cpu potrebbe essere impiegato per eseguire la visita di un sottoalbero non ancora assegnato [MarCam82]. Questa tecnica è chiarita da Fig. 4.22.

Diversi studiosi hanno cercato di migliorare questo algoritmo di base. L'approccio mandatory work first dovuto ad Akl, Barnard e Doran ne è un esempio [AkBaDo82].

4.2.5.4 Mandatory work first

Mandatory work first è un metodo che impiega un pool di processori in una ricerca parallela condotta in due fasi.

Nel primo stadio della ricerca è esplorata soltanto una porzione ben definita dell'albero di gioco: l'albero minimale. Nella seconda visita è esaminato il resto dell'albero di gioco usando i limiti α e β trovati durante la prima ricerca.

Durante le due fasi della ricerca l'algoritmo gestisce una lista di sottoalberi da visitare stabilita da esplorazioni precedenti. I processori prelevano un sottoalbero dalla lista, lo visitano, comunicano il risultato e talvolta aggiungono lavoro alla lista.

L'algoritmo opera una classificazione dei figli di ciascun nodo. Il primo figlio di un nodo è chiamato figlio sinistro. Il sottoalbero contenente questo figlio è detto sottoalbero sinistro ed è esplorato da un processo di tipo S. Tutti gli altri figli di un nodo sono chiamati figli destri e sono contenuti nei sottoalberi destri visitati da processi di tipo D.

Dato un nodo il sottoalbero sinistro viene esplorato da un processo S finché non è fatto risalire al nodo il valore del figlio sinistro. Per ottenere questo scopo il processo S genera processi (di tipo sia S che D) per esplorare tutti

i sottoalberi del figlio sinistro. Parallelamente, per ognuno dei figli destri è ricavato un valore temporaneo.

Questi valori sono confrontati con il valore finale del figlio sinistro per produrre eventuali tagli. Questo valore temporaneo è calcolato da un processo D, il quale genera processi S e D per valutare il sottoalbero sinistro del figlio destro cui è associato. Se non è generato un taglio la visita del sottoalbero destro continua generando un processo D per esplorare il secondo sottoalbero del figlio destro. Tale processo viene arrestato quando il sottoalbero destro è stato completamente esplorato o si ha un taglio.

L'algoritmo di Fig. 4.23 illustra questo evolversi di attivazione e terminazione di processi S e D.

```
/* l'algoritmo ricorda fra parentesi angolate
(< e >) i punti dove inserire le attività per
la gestione dei valori minimax intermedi e
finali */
typedef tuple_name ...;
int mandatory_work_first(position *p,int
depth)
{
tuple_name roothandled,lsondone;
void handle(position
*,int,int,int,tuple_name,tuple_name);
handle(*p,depth,TRUE,FALSE,roothandled,lsondon
e);
in(roothandled);
<return minimax value>;
}
void handle(position p,int depth,int left,int
parentleft,
tuple_name done,tuple_name lsiblingdone)
{
tuple_name sondone,lsdone;
position *successor;
int nmoves,cutoff,i;
if (depth==0)
       {
       < valuta posizione >;
       if (!left && parentleft)
              in(lsiblingdone);
       }
else
       {
       nmoves=genmoves(p,&successor);
       if (left)
              {

       for(i=1;i<=nmoves;i++,successor++)

       eval("split",handle(successor,depth-
1,i==0,left,sondone,lsondone));
              for (i=1;i<=nmoves;i++)
                     in(sondone);
              }
       else
              {
```

```
                cutoff=false;
                for (i=1;i<=nmoves &&
!cutoff;i++,successor++)
                        {

        handle(successor,i==1,left,sondone,lson
done);
                        in(lsiblingdone);
                        out(lsiblingdone);
                        cutoff=<il mio valore
temporaneo è peggiore di quello di mio padre>;
                        }
                }
        }
< modifica, se migliorato, il valore di mio
padre >;
if (left && leftparent)
        out(lsiblingdone);
out(done);
}
```

Fig. 4.23 Mandatory work first

Il vantaggio di questo algoritmo è la riduzione dell'overhead di ricerca in quanto è eseguito solo il lavoro che è stato dimostrato essere necessario.

Nonostante questo metodo proponga la possibilità di prestazioni ottime (in termini di complessità in tempo), il suo utilizzo reale è problematico a causa dell'enorme spazio di memoria necessario per memorizzare tutte le valutazioni parziali dei nodi dell'albero [AkBaDo82].

4.2.5.5 PVSplit (Principal Variation Splitting)

PVSplit è un algoritmo parallelo di ricerca $\alpha\beta$ molto efficiente nella visita di alberi di gioco fortemente ordinati [MarCam82;MarPop85].

Per comprendere l'idea del metodo è utile esaminare la natura dell'albero visitato da un algoritmo $\alpha\beta$ in condizioni ottime di ordinamento. Abbiamo già discusso della classificazione dei nodi di un albero di gioco in 3 tipi: PV, CUT e ALL.

L'efficacia della ricerca $\alpha\beta$ deriva dal fatto che i nodi CUT possono essere tagliati prima che l'albero relativo sia completamente esplorato. Il massimo beneficio da questo taglio si ottiene quando il miglior valore per il parametro A è disponibile. L'idea del metodo è di cercare di trovare questo valore prima che inizi l'esplorazione dei nodi CUT.

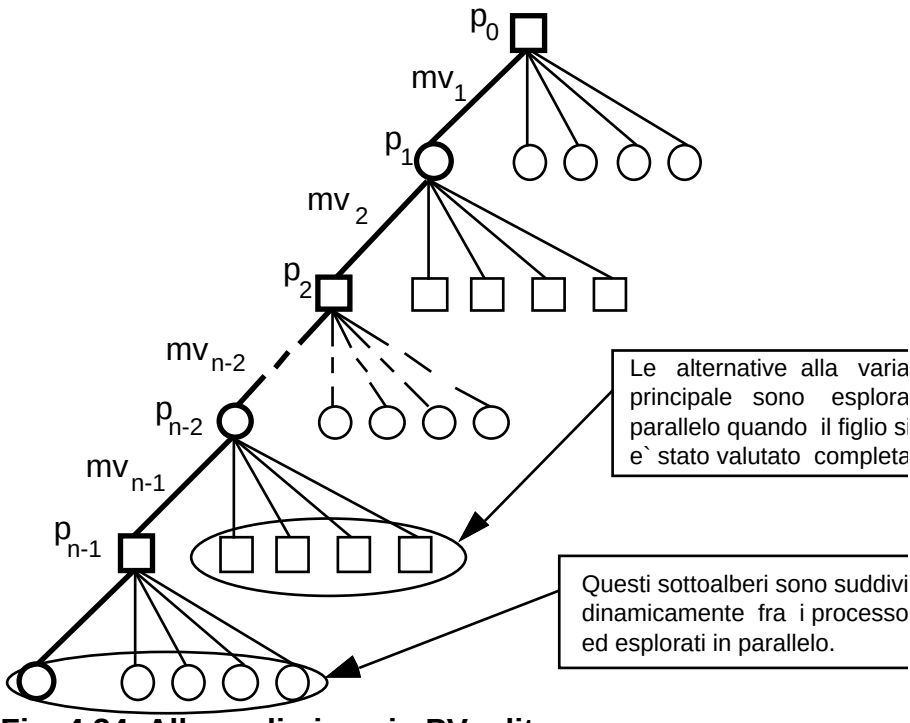


Fig. 4.24 Albero di gioco in PVsplit

Inizialmente tutti i processori sono impiegati nella ricerca del sottoalbero più a sinistra del nodo radice, dove si assume sia contenuta la principal variation. In questo modo la mossa più promettente è valutata molto velocemente e la successiva decomposizione della restante parte dell'albero di gioco vede i processori iniziare la loro visita con una buona finestra di ricerca. Ciò consente più tagli ed un minore overhead di ricerca rispetto all'algoritmo base di decomposizione.

```
#define update_alpha(VALUE)\
{\
if (VALUE>alpha)\
       alpha=VALUE;\
if (alpha>=beta)\
       {\
       terminate(); /* implementa la
terminazione di tutti i sottoprocessori */ \
       return(alpha);\
       }\
}\
int pvsplit (position *p,int alpha,int
beta,int length)
{
position *successor;
int nmoves,result,i;
if (length==0)
       return (treesplit(*p,alpha,beta));
nmoves=genmoves(p,&successor);
if (nmoves==0)
       return(evaluate(p));
alpha=-pvsplit (successor,-beta,-alpha,length-
1);
```

```
if (alpha>=beta)
       return(alpha);
for (ind=1,free=NSONS;i<nmoves;ind++)
       {
       eval
("pvsplit",treesplit(*(successor+ind),-beta,-
alpha,length-1));
       free--;
       if (!free)
              {
              in("pvsplit",?result);
              free++;
              update_alpha(-result);
              }
       }
while (free<NSONS)
       {
       in ("pvsplit",?result);
       free++;
       update_alpha(-result);
       }
return (alpha);
}
```

Fig. 4.25 PVSplit

Analizziamo questo algoritmo più in dettaglio. PVSplit esegue una sequenza di ricerche secondo l'euristica di approfondimento iterativo.

Sia mv_1, ..., mv_{n-1} la continuazione principale trovata nella (n-1)_esima iterazione e siano p_0, p_1, ..., p_{n-1} i nodi corrispondenti, dove p_0 è il nodo radice e p_{n-1} la posizione raggiunta al termine della sequenza (Fig. 4.24). Questi nodi sono chiamati nodi splitting.

Quando la n-esima iterazione ha inizio tutti i processori sono impiegati nella valutazione (secondo l'algoritmo di decomposizione di base) dell'albero avente come radice p_{n-1}.

Quando tale ricerca è completata il valore minimax di p_{n-1} risale fino a p_{n-2} e le rimanenti mosse legali in p_{n-2} sono suddivise dinamicamente fra i processori ed esplorate in parallelo.

Terminata la valutazione di p_{n-2}, il suo valore è fatto risalire fino a p_{n-3} e così via. Tale processo è ripetuto per ogni nodo splitting fino a che le mosse nella radice sono finalmente suddivise fra i processori per produrre il valore minimax finale ed una nuova principal variation per la succesiva iterazione.

In Fig. 4.25 è presentata una versione C-Linda di PVSplit.

4.2.5.6 Varianti dell'algoritmo PVSplit

L’overhead di sincronizzazione è il motivo principale delle prestazioni non soddisfacenti di PVSplit.

La soluzione proposta da Schaeffer è di riutilizzare processori inattivi per "assistere" quelli attivi. La sua idea ha originato l’algoritmo DPVSplit (Dynamic PVSplit) [Sch89].

PVSplit è basato su un’organizzazione statica della gerarchia di processori ed impedisce così che i nodi foglia di questa gerarchia possano rendere divisibile il lavoro assegnato ed usufruire dell’aiuto di altri processori.

In DPVSplit è proposta l’idea di una struttura ad albero dinamica per i processori. Esiste un processo supervisore ed un insieme di esploratori. Ogni esploratore riceve dal supervisore un lavoro (divisibile) ed applica PVSplit al sottoalbero corrispondente a questo lavoro. Quando il supervisore non ha più sottoalberi da distribuire, cerca un esploratore E_1 inattivo per aiutare un esploratore E_2 nel completare la sua ricerca. E_1 dovrà rispondere a E_2 del risultato della ricerca.

DPVSplit riduce sensibilmente l’overhead di sincronizzazione, ma non completamente: un processore che ha assegnato lavoro a certi esploratori deve attendere che tutti questi abbiano terminato prima di iniziare una nuova ricerca.

Il metodo introduce, rispetto a PVSplit, un maggior numero di comunicazioni a causa della maggiore cooperazione tra i processori. È maggiore anche l’overhead di ricerca in quanto dalla frammentazione dei lavori risulta una perdita di informazioni. Infatti se un sottoalbero è visitato dallo stesso processore le tavole history e delle trasposizioni possono essere utilizzate col massimo beneficio, ma se parte del sottoalbero è esplorata su un altro processore molte informazioni non sono reperibili nelle tavole locali e non possono originare certi tagli.

Vi sono due aspetti di PVSplit che hanno suggerito a Newborn l’idea per il progetto di un nuovo algoritmo parallelo [New88].

La filosofia di PVSplit è di far risalire molto velocemente verso ogni nodo splitting la valutazione del nodo splitting successivo, così

che questo valore possa essere usato da tutti i processori quando essi intraprendono l'analisi delle rimanenti mosse in quel nodo.

Una seconda caratteristica di PVSplit è che tutti i processori devono aver completato la propria ricerca nella iterazione corrente prima che la successiva inizi: quei processori che terminano la ricerca prima di altri restano inattivi in attesa che questi finiscano.

L'algoritmo UIDPABS (Unsynchronized Iteratively Deepening Parallel Alpha Beta Search) è una variante di PVSplit che non presenta le due caratteristiche ricordate [New88].

Questo metodo di ricerca procede seguendo i passi:

1. durante le prime due iterazioni tutti i processori fanno le stesse ricerche per determinare un efficace ordinamento delle mosse generate nella radice. Il valore finale ottenuto dalla seconda iterazione è usato come centro di una finestra $\alpha\beta$ utilizzata nelle successive iterazioni. La possibilità di disporre di una buona finestra $\alpha\beta$ riduce l'urgenza di far risalire la valutazione dei nodi splitting verso la radice.

2. le mosse alla radice sono ripartite uniformemente fra i processori

3. ogni processore intraprende una ricerca iterativa sull'albero definito dal proprio sottoinsieme di mosse con la finestra iniziale definita in 1.

La ricerca è indipendente fra i processori, anche rispetto a iterazioni successive: un processore può trovarsi nel mezzo della 6ª iterazione, mentre un altro è ancora alle prese con la 5ª

4. se la ricerca di un processore fallisce la finestra $\alpha\beta$ viene traslata verso l'alto o verso il basso a secondo che il fallimento sia superiore o inferiore. Inizia così una nuova ricerca. Di fronte ad un nuovo fallimento, tuttavia, l'esplorazione sarà ritentata con una finestra $\alpha\beta$ tale da contenere sicuramente la soluzione

5. la ricerca globale ha termine quando è raggiunto un certo tempo limite di esecuzione.

UIDPABS dimostra che due caratteristiche fondamentali di PVSplit (sincronizzazione fra

iterazioni successive e veloce valutazione della variante principale) non sono requisiti necessari di un algoritmo parallelo efficiente.

4.3 Effetto delle euristiche nella ricerca parallela

Nella implementazione delle soluzioni parallele di ricerca $\alpha\beta$ si è continuato a fare uso di euristiche con l'intento di migliorarne l'efficienza. L'efficacia di queste euristiche è rimasta inalterata nel passaggio da ambiente sequenziale a parallelo?

Le tabelle di trasposizione continuano ad essere pienamente efficaci, a patto che tutti i processori accedano alla stessa tabella globale. Essendo tuttavia consultata molto frequentemente essa diviene un collo di bottiglia per le richieste dei processori causando così un aumento degli overheads di sincronizzazione e comunicazione.

Un processore potrebbe fare qualcosa di utile mentre è in attesa della risposta alla sua interrogazione; ad esempio iniziare a visitare il successivo sottoalbero. Se l'informazione richiesta non è nella tabella allora non si è perso del tempo, altrimenti è utilizzato il primo pervenuto fra i risultati della ricomputazione del sottoalbero e la risposta dalla tabella.

Una soluzione che riduce i tempi di accesso alla tabela delle trasposizioni è di suddividere la stessa in diverse partizioni ed associare ciascuna ad un processore. La tabella rimane condivisa fra i processori, ma è minore la probabilità che due di essi debbano sincronizzarsi sulla stessa porzione. La gestione della tabella diviene però più complessa e normalmente un processore è dedicato a definire e controllare il routing delle interrogazioni, anche se talvolta sono gli stessi processori a conoscere quale fra di essi dispone dell'informazione cercata.

Esiste la possibilità che ogni processore mantenga una propria tabella locale di trasposizioni. Questa soluzione aumenta l'overhead di ricerca in quanto può accadere che un processore valuti un sottoalbero già esplorato da un altro; offre però il vantaggio di non ritardare l'accesso alla tabella.

La preferenza della soluzione centralizzata rispetto alla distribuita è fortemente dipendente dalle dimensioni della tabella e dalle connessioni fra i processori.

L'implementazione delle altre basi di conoscenza, come la tabella history, la lista dei killer e la tabella delle confutazioni presenta problemi analoghi a quelli appena visti.

L'efficacia dei meccanismi di ordinamento delle mosse è legata al tipo di algoritmo di ricerca.

Ad esempio essi sono poco utili nell'algoritmo di decomposizione di base. I programmi sequenziali usano il risultato della valutazione del primo sottoalbero per tagliare grossa parte del resto dell'albero. Se tutti i sottoalberi di un

nodo sono esplorati in parallelo, questo risultato non è disponibile quando inizia la valutazione degli altri sottoalberi.

Viceversa in algoritmi paralleli che sono efficienti nella ricerca di alberi fortemente ordinati (PVSplit), i meccanismi di ordinamento sono essenziali.

Sono stati già discussi i problemi introdotti in algoritmi distribuiti dall'utilizzo della tecnica di approfondimento iterativo: si ha un incremento dell'overhead di sincronizzazione perché tutti i processori devono attendere il completamento della corrente iterazione prima di affrontare la successiva. È già stata analizzata una soluzione a questo inconveniente che però introduce una più complessa strategia di schedulazione dei lavori.

Capitolo 5
Algoritmi di decomposizione dell’albero di gioco

5.1 Introduzione

Nell’ambito della presente discussione per distribuzione della ricerca si deve intendere l’impiego di una molteplicità di risorse di elaborazione (processori) nel portare a termine la visita di un albero di gioco in accordo con un generico algoritmo di ricerca. Oggetto di questa fase del lavoro è proprio lo studio e la verifica sperimentale di alcuni metodi di distribuzione della ricerca.

Analisi e sperimentazione di tali metodi verranno sviluppate isolando e minimizzando ogni dipendenza dal dominio di applicazione. In particolare ogni processore impiegato nella ricerca distribuita fruirà della stessa conoscenza del dominio, rendendo così ciascuno di essi indistinguibile sotto questo punto di vista.

Nel Capitolo 4 è stata presentata una classificazione dei metodi di distribuzione della ricerca su alberi di gioco unitamente ad una rassegna dei più significativi; già in quella sede furono enfatizzate importanza ed efficacia di un particolare insieme di algoritmi distribuiti di ricerca: quelli detti di decomposizione dell’albero di gioco. L’attenzione sarà focalizzata proprio su questa classe di algoritmi. Quali le motivazioni di questa scelta?

Gli algoritmi di decomposizione sono stati oggetto di esaurienti studi e sperimentazioni per quanto concerne, almeno, l’implementazione su architetture con limitato parallelismo (nr. di processori ≤ 12). L’affacciarsi di architetture a parallelismo massiccio ha nuovamente riaperto la discussione riguardo le prestazioni dei diversi algoritmi di decomposizione; tuttavia alcuni risultati sperimentali appaiono confermare la bontà di soluzioni algoritmiche risultate già "vincenti" in presenza di un numero esiguo di processori [FeMyMo90; FMMV89]. Non potrebbe dunque apparire ozioso un nuovo lavoro di analisi e sperimentazione di tali algoritmi su un’architettura a limitato parallelismo quale una rete di workstation?

Deve essere chiaro che le finalità di questo stadio del lavoro esulano dallo sviluppo di un nuovo algoritmo distribuito di visita di alberi di gioco avente prestazioni confrontabili o superiori a quelle dei migliori già individuati. L’obbiettivo che si intende conseguire è invece un esame quanto più completo del modello di comunicazione basato su spazi di tuple, condotto rispetto a parametri di valutazione quali la facilità di programmazione e l’efficienza. In quest’ottica gli algoritmi di decomposizione si rivelano uno strumento di test "completo". Al variare del criterio di decomposizione, infatti, essi divengono rappresentativi di uno spettro molto esteso di problemi, con granularità del parallelismo

che va dalla grana grossa alla grana fine. La scelta di concentrare gli sforzi su una classe di algoritmi abbondantemente sperimentati va inoltre inquadrata nell'esigenza di disporre di un numero sufficiente di risultati sperimentali ottenuti su diverse architetture hardware e software, tali così da costituire essenziali termini di confronto per i risultati che saranno presentati in questo lavoro.

Il linguaggio Linda concretizza un modello di programmazione parallela relativamente nuovo e quindi sperimentato in un numero limitato di domini (fra i primi si ricordano moltiplicazione di matrici e decomposizione LU [CaGeLe88], confronto di sequenze di DNA e database con ricerca parallela [CarGel88a], algoritmi branch-and-bound e sistemi esperti [FacGel88], programmi di ray-tracing [LeiWhi88]). L'area dei problemi di ricerca guidata da euristiche si propone pressoché inesplorata dalla programmazione in Linda; in quest'area i metodi di decomposizione dell'albero di gioco costituiscono esempi di notevole complessità, così da rivelare di suggestivo interesse la loro sperimentazione in Linda.

Questa fase del lavoro si svilupperà dunque attraverso l'analisi e l'implementazione di alcuni algoritmi di decomposizione dell'albero di gioco. L'esplorazione di tali algoritmi procederà in modo incrementale rispetto alla complessità delle interazioni che essi inducono fra i processori. Inizialmente saranno quindi presentati metodi di decomposizione con minima cooperazione; successivamente verranno esaminate forme di interazione fra processori via via più complesse. Questo tipo di approccio si presenta di notevole interesse data la natura degli algoritmi paralleli in analisi. Essi nascono come generalizzazione nel distribuito dell'algoritmo sequenziale $\alpha\beta$. La visita in parallelo di porzioni disgiunte dell'albero di gioco origina overhead di ricerca rispetto all'algoritmo sequenziale; questa forma di overhead contribuisce al degrado delle prestazioni. Il suo manifestarsi deriva dal fatto che i processori non hanno una visione globale di informazioni da essi ricavate durante la visita di rispettive porzioni dell'albero. I metodi di decomposizione con forme di cooperazione più complesse sono finalizzati, generalmente, alla riduzione dell'overhead di ricerca attraverso una maggiore condivisione di queste informazioni. Per contro, un aumento delle interazioni determina un maggior numero di comunicazioni e punti di sincronizzazione, motivi anche questi di degrado delle prestazioni. Sarà di primario interesse verificare fino a quale grado di condivisione i vantaggi che derivano dalla riduzione dell'overhead di ricerca non sono vanificati dall'aumento del degradi legati alla comunicazione ed alla sincronizzazione.

Sono necessarie analoghe considerazioni riguardo quei metodi di decomposizione finalizzati al bilanciamento del carico fra i processori e quindi alla riduzione del degrado legato alla sincronizzazione. Questa classe di metodi, infatti, origina un aumento delle comunicazioni e, più importante, dell'overhead di ricerca. Si può dunque osservare che la determinazione di un

algoritmo distribuito di ricerca efficiente deriva da soluzioni di compromesso (trade-off) che minimizzino complessivamente tutte le forme di degrado delle prestazioni. Le considerazioni sin qui formulate fanno presagire l'impossibilità di esistere di un algoritmo di decomposizione universalmente ottimo, le cui prestazioni, cioè, non dipendano dagli strumenti software impiegati nella programmazione delle interazioni e soprattutto dall'architettura hardware parallela sulla quale l'algoritmo è implementato.

Questa realtà, peraltro comune a molte classi di algoritmi paralleli, conferisce un ulteriore significato al presente lavoro; esso può essere interpretato come esempio di metodologia di ricerca guidata dall'ambiente di programmazione e di sperimentazione.

L'indagine sperimentale affronterà inizialmente i metodi di decomposizione statica e successivamente quelli di natura dinamica. Nel corso del Capitolo sarà inoltre presentata la sperimentazione nel distribuito di alcune euristiche di ordinamento dei nodi interni dell'albero di gioco.

5.2 Algoritmi di decomposizione dell'albero di gioco

Si consideri l'algoritmo sequenziale $\alpha\beta$ (Fig. 1.9). Esso descrive la visita di un albero di gioco in ordine depth-first, sviluppata fino ad una profondità finita e finalizzata al calcolo del valore minimax del nodo alla radice dell'albero. Gli algoritmi di decomposizione costituiscono una generalizzazione in ambiente distribuito dell'algoritmo $\alpha\beta$. Essi, infatti, definiscono una classe di metodi di distribuzione della visita da esso indotta; la caratteristica di questo approccio è che la ricerca su sottoalberi distinti dell'albero di gioco è a carico di processori in generale diversi e quindi condotta simultaneamente.

Gli algoritmi di decomposizione possono essere partizionati in due sottoclassi:

- algoritmi con decomposizione statica
- algoritmi con decomposizione dinamica.

La natura statica della decomposizione deriva, per il primo tipo di algoritmi, dal fatto che i nodi di decomposizione sono fissati a priori, cioè prima che la visita dell'albero di gioco abbia inizio.

Gli algoritmi di decomposizione dinamica, invece, prevedono che un nodo interno venga designato o meno come di decomposizione solamente al momento della visita dell'albero.

5.2.1 Algoritmi di decomposizione statica

5.2.1.1 L'algoritmo base di decomposizione

In 4.2.5.3 è stato presentato un algoritmo di decomposizione definito di base, data la sua semplicità. Tale algoritmo introduce un'architettura logica di comunicazione denominata albero dei processori. Si consideri la versione semplificata di tale algoritmo in cui l'albero dei processori ha profondità 1 (Fig. 5.1).

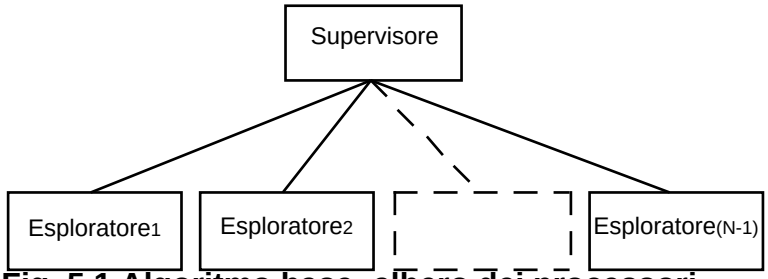


Fig. 5.1 Algoritmo base: albero dei processori

Qual è il significato di questo schema di comunicazione?

Dato un insieme di N processori, solamente uno di essi è supervisore, mentre i restanti N-1 sono esploratori. Il processore supervisore è il responsabile del nodo radice e quindi della ricerca dell'intero albero di gioco. Il suo compito è distribuire ai suoi sottoprocessori la visita dei sottoalberi della radice. La particolare struttura dell'albero dei processori impedisce qualsiasi forma di subappalto della ricerca: gli esploratori, divenuti su incarico del supervisore responsabili di uno dei sottoalberi della radice, devono completare sequenzialmente la visita di quest'albero non potendo disporre della collaborazione di propri sottoprocessori. L'algoritmo di decomposizione che emerge da questo approccio induce la presenza di un unico nodo di decomposizione: la radice dell'albero di gioco (Fig. 5.2). La distribuzione della ricerca è dunque confinata unicamente al top-level dell'albero, dove è stabilito un numero di ricerche sequenziali pari ai successori della radice.

È evidente la relazione gerarchica stabilita fra il processore supervisore e tutti gli esploratori: esso specifica quali sottoalberi della radice devono essere visitati, mentre gli esploratori debbono completare tali visite e rispondere al supervisore del loro operato comunicandone il risultato. Quest'ultimo gestisce dinamicamente (cioè durante la ricerca) tali risultati, così da "migliorare" la ricerca[20] dei sottoalberi non ancora distribuiti.

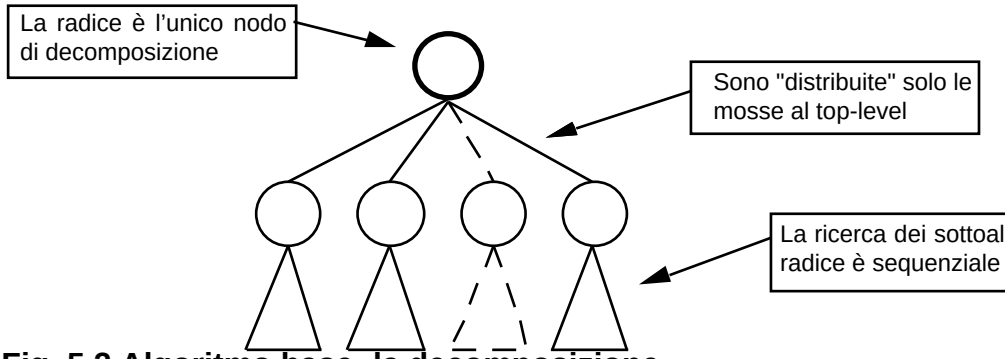


Fig. 5.2 Algoritmo base: la decomposizione dell'albero di gioco

---

È dinamica anche l'attribuzione agli esploratori della visita dei sottoalberi. Il supervisore non ha interesse a conoscere quale degli esploratori visiterà un certo sottoalbero: ciò che per esso è importante è che siano completate tutte le visite richieste e non l'identità degli agenti che le hanno prodotte. Gli esploratori appaiono quindi al supervisore come indistinguibili data, inoltre, la loro omogeneità: tutti svolgono la stessa attività, cioè la ricerca sequenziale $\alpha\beta$. La comunicazione da parte del master della richiesta di ricerca di un nuovo sottoalbero è quindi di tipo asimmetrico in uscita: è rivolta a tutti gli esploratori, ma potrà essere "consumata" solamente dal primo di essi che la riceverà.

Le interazioni descritte fanno emergere un'architettura logica di comunicazione perfettamente assimilabile al modello master-worker di programmazione parallela (Fig. 5.3). Esiste infatti un'analogia perfetta fra agenti ed azioni dell'algoritmo in esame e quelli descritti nella definizione del modello (cfr. 3.2.2.4.1): un processo master (supervisore) genera un insieme ordinato (nel presente caso l'ordine è casuale) di lavori (ricerca $\alpha\beta$ di sottoalberi della radice); l'esecuzione dei lavori è ad opera di un insieme di processi worker omogenei ed indistinguibili (esploratori) i quali comunicano il risultato del rispettivo lavoro (valore minimax del sottoalbero visitato) al master cui ne compete l'elaborazione (aggiornamento del valore minimax dell'albero e della finestra $\alpha\beta$ da utilizzare nei successivi lavori).

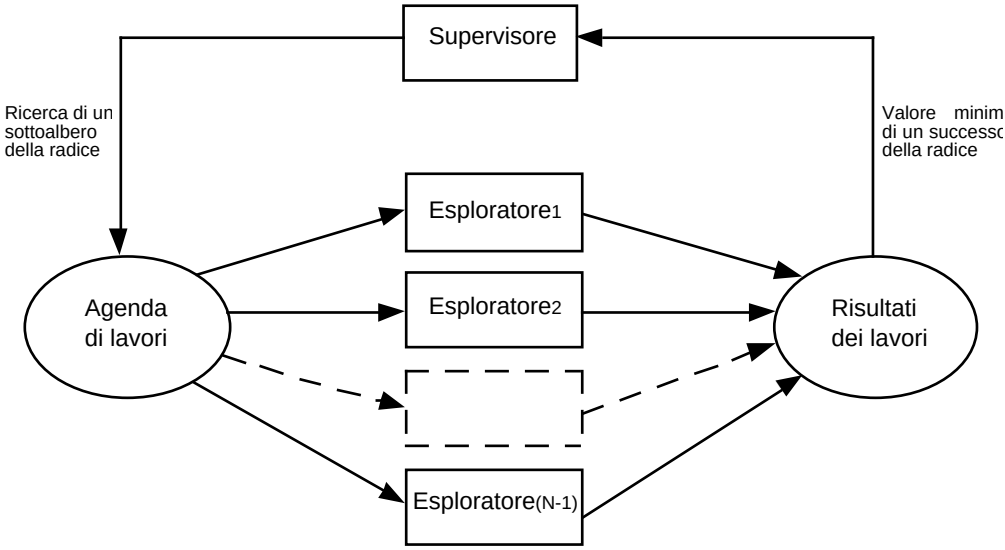


Fig. 5.3 Algoritmo base: architettura logica di comunicazione

---

5.2.1.1.1 Il progetto della cooperazione attraverso lo spazio delle tuple

Il linguaggio Linda costituisce uno strumento estremamente espressivo nella programmazione di interazioni fra processi basate sul modello master-worker. Il motivo è che in generale è agevole ed immediata la definizione come insieme di tuple delle strutture:

• agenda, cioè della struttura atta a conservare (ed eventualmente ordinare) la lista di lavori ancora inevasi

• insieme dei risultati prodotti dal lavoro dei worker.

In riferimento all'algoritmo di decomposizione in esame, entrambe queste strutture sono implementate come strutture dati distribuite di tipo bag.

L'agenda di lavori è implementata da tuple il cui schema è:

```
("job", subtree)
```

Il campo subtree identifica univocamente uno dei sottoalberi della radice. In un albero di gioco deterministico un suo sottoalbero è individuato dal rispettivo nodo radice. Tale nodo, come qualsiasi altro, rappresenta uno stato del gioco; in generale questa informazione è di grosse dimensioni (si pensi alla rappresentazione della posizione dei pezzi negli scacchi) e quindi il costo della sua

comunicazione può risultare molto elevato. Data la particolarità dell'algoritmo di base è possibile un'ottimizzazione per cui il tipo di questo campo può essere ridotto ad un intero. Si osservi infatti che:

• è nota staticamente l'identità dell'unico nodo di decomposizione

• è realistica l'assunzione che, dato uno stato del gioco, il generatore di mosse di ogni processore generi quest'ultime nel medesimo ordine.

Grazie a queste particolarità dell'algoritmo di base il campo subtree conterrà unicamente l'indice ordinale della mossa al top-level che ha originato il sottoalbero che deve essere visitato. Al fine di garantire la correttezza di questo approccio la ricerca complessiva deve essere preceduta da una fase di inizializzazione in cui ogni esploratore:

• riceve dal supervisore la posizione alla radice dell'albero di gioco (anche per questa interazione è possibile l'ottimizzazione in cui viene comunicato l'indice della mossa realmente giocata) e

• genera autonomamente la sequenza di mosse al top-level.

La completa specifica della visita $\alpha\beta$ di un albero di gioco si compone, oltre che del nodo radice, anche dei limiti della finestra (alpha, beta) e della profondità nominale di ricerca. Quest'ultima informazione è costante e viene quindi comunicata per diffusione a tutti i processori prima che la ricerca abbia inizio. Al top-level la finestra (alpha, beta), invece, viene aggiornata dinamicamente dai risultati ritornati dagli esploratori; i suoi limiti devono quindi costituire parte integrante della specifica di un lavoro. Perché questi dati non sono allora contenuti nei campi delle tuple che definiscono l'agenda?

Si considerino a riguardo le seguenti osservazioni:

• si supponga che l'algoritmo non sia inserito in una ricerca di tipo aspiration[21]; questa ipotesi ha come implicazione che la finestra iniziale $\alpha\beta$ al top-level sia: $(-\infty,+\infty)$. Il limite superiore $+\infty$ è costante durante l'intera ricerca e quindi già

noto agli esploratori; lo score (così è anche chiamato il limite inferiore α), invece, può aumentare dinamicamente. Esso rappresenta il valore minimax associato alla migliore fra le mosse già esplorate; il suo aggiornamento è a cura del supervisore ed ha luogo qualora un esploratore ritorni una valutazione per la mossa assegnatagli maggiore dello score corrente, divenendo così essa il nuovo score. Il valore attuale dello score è quindi noto solamente al supervisore; un generico lavoro dovrà quindi includere logicamente questa informazione.

• come accennato, una possibile soluzione è quella di inserire lo score corrente in un campo aggiuntivo della tupla-lavoro. Questo tipo di approccio si rivela però insoddisfacente in prospettiva di una futura "evoluzione" dell'algoritmo. Si consideri, ad esempio, il seguente scenario: il supervisore ha inserito in agenda un nuovo lavoro L contenente il sottoalbero P ed il valore S_1 per lo score corrente; successivamente un esploratore termina la sua ricerca provocando un miglioramento dello score che assume ora valore S_2. Supponendo che il lavoro L non sia stato ancora rimosso dall'agenda, si osservi come esso contenga un valore per lo score non aggiornato. L'aggiornamento di questo campo è certamente inefficiente, in quanto richiede rimozione ed inserimento anche delle informazioni di specifica del sottoalbero (rimaste comunque inalterate). Tale situazione si verifica tanto più frequentemente quanto maggiore è il tempo medio di permanenza in agenda dei lavori; nel caso dell'algoritmo base, tuttavia, tale tempo è molto basso rendendo così impercettibile il fenomeno descritto (il motivo sarà spiegato fra breve).

• soluzione alternativa è quella di separare fisicamente le informazioni relative al sottoalbero e allo score. Si osservi come quest'ultimo non costituisca elemento di distinzione fra i lavori; lavori generati consecutivamente conterranno con molta probabilità lo stesso score (a meno di un suo aggiornamento nel tempo intermedio fra le due generazioni). Ai fini della correttezza

dell’algoritmo tale informazione non deve essere necessariamente trasmessa; la sua finalità è unicamente velocizzare la ricerca. Ciò che è importante è stabilire una forma di condivisione dello score fra supervisore ed esploratori. Ogni soluzione ragionevole a questo problema deve prevedere che ciascun esploratore mantenga una copia locale del valore dello score e che questa venga aggiornata periodicamente; il fatto che tale aggiornamento abbia luogo al momento della ricezione di un nuovo lavoro è solo una fra le possibili scelte.

L’approccio che emerge dalla presente discussione è l’implementazione della condivisione dello score mediante una struttura dati del tipo variabile distribuita; essa sarà caratterizzata da un’unica tupla con struttura

```
("score", valore)
```

aggiornata dal supervisore ed acceduta in sola lettura dagli esploratori. Questa soluzione gode di interessanti proprietà che offrono lo spunto per alcune riflessioni di carattere generale:

• contrariamente alla prima soluzione presentata, ora il supervisore partecipa attivamente alla condivisione dello score solamente quando è necessario, cioè quando il suo valore deve essere aggiornato; i riflessi sull’efficienza sono evidenti: nel caso di un albero ordinato lo score al top-level viene modificato molto raramente poiché i suoi miglioramenti saranno prodotti con molta probabilità solamente dalle prime mosse al top-level.

• la soluzione permette di applicare con maggiore grado di libertà uno dei criteri cardine della programmazione parallela: il principio di autonomia dei moduli. Essa offre infatti ai moduli esploratori la possibilità di decidere autonomamente quando aggiornare la copia locale dello score; il rispetto del principio di autonomia è generalmente origine di una struttura di interazioni flessibile che nel caso della presente discussione significa poter sperimentare facilmente frequenze diverse di aggiornamento locale dello score (ad es. basate su intervalli di tempo regolari in alternativa alla frequenza di aggiornamento

scandita dall’evento "ricezione di un nuovo lavoro").

• la condivisione di una variabile distribuita è causa di serializzazione fra i processori; il degrado prodotto da tale fenomeno è proporzionale alla durata media della relativa sezione critica (cioè del tempo di accesso alla variabile) ed alla frequenza con cui esso si manifesta (nella fattispecie pari alla frequenza dell’operazione di aggiornamento). L’incidenza di questo degrado sull’algoritmo di base è minima poiché la quasi totalità degli accessi alla variabile "score" è in lettura e la frequenza relativa degli aggiornamenti è, come spiegato, molto limitata.

È finalmente definita un’agenda in cui i lavori sono inseriti dal supervisore e prelevati dagli esploratori.

La ricerca $\alpha\beta$ trae beneficio dal fatto che le mosse "migliori" (con maggiore valore minimax) siano valutate per prime; è per questo motivo che le mosse prodotte dall’omonimo generatore sono successivamente ordinate in base ad una loro valutazione statica. Il supervisore inserisce i lavori nell’agenda nello stesso ordine delle mosse al top-level cui essi si riferiscono. L’organizzazione dell’agenda in una struttura di tipo bag rende però i lavori indistinguibili; ciò implica che l’ordine di estrazione dei lavori non rispetti in generale l’ordine cronologico di inserimento. Perché, dunque, non implementare l’agenda come una struttura dati distribuita più complessa (ad es. coda-in) che permetta di preservare l’ordinamento dei lavori al momento dell’estrazione?

Questo tipo di soluzione si rivelerà necessario in algoritmi più complessi onde limitare l’overhead di ricerca. Tuttavia l’algoritmo di base gode di una proprietà per cui la rinuncia all’ordinamento dei lavori non determina un reale aumento dell’overhead di ricerca offrendo così il vantaggio di disporre di un’agenda di lavori con struttura semplice e con accesso ad essa più efficiente. Tale proprietà consiste nel fatto che il supervisore conosce, in ogni istante della ricerca, il numero di esploratori inattivi. Di conseguenza, esso inserisce un nuovo lavoro

in agenda solamente se almeno un esploratore è già pronto a raccoglierlo; la trasmissione di un lavoro viene quindi ritardata fino al momento in cui è possibile la sua immediata esecuzione. Da un punto di vista logico tutti i lavori presenti contemporaneamente in agenda vengono dunque rimossi simultaneamente, rendendo così superflua una loro gestione ordinata.

Il fatto che il supervisore generi un lavoro solo se almeno un esploratore è libero costituisce inoltre il motivo per cui il tempo medio di permanenza dei lavori in agenda è pressoché trascurabile.

Anche l'insieme di risultati prodotti dai moduli esploratori sono organizzati in una struttura dati distribuita di tipo bag. Le tuple che definiscono questa struttura hanno il seguente schema:

```
("result", value, subtree)
```

Il campo value contiene il valore minimax per il sottoalbero della radice identificato dal contenuto del campo subtree. La presenza di quest'ultima informazione non è necessaria ai fini del calcolo del valore minimax complessivo, ma è tuttavia indispensabile per la determinazione della migliore mossa.

La decisione precedentemente operata di non gestire l'ordinamento dei lavori in agenda rende impossibile stabilire qualche ordinamento significativo per la struttura di risultati. Quali vantaggi avrebbe comportato la scelta di recuperare i risultati nello stesso ordine con cui i rispettivi lavori sono stati generati ed inseriti in agenda?

Si consideri la ricerca di un albero di gioco perfettamente ordinato: in questo caso la lettura ordinata dei risultati fa sì che lo score venga aggiornato una sola volta (dal risultato della visita della prima mossa); in tale albero, infatti, essendo la prima mossa anche la migliore, il suo valore non potrà essere migliorato da nessuna delle successive visite. Un primo vantaggio che quindi emerge per questo approccio è una limitazione del numero di aggiornamenti dello score. Si osservi inoltre che elaborare per primi i risultati delle mosse presunte migliori consente in generale di

ottenere uno score parziale più elevato e quindi un maggiore restringimento della finestra αβ utilizzata nei successivi lavori con conseguente riduzione dell'overhead di ricerca.

Va considerato, tuttavia, che il tempo di completamento dei lavori è impredicibile data la natura della ricerca αβ per cui la ricerca di due alberi delle stesse dimensioni può richiedere la visita di insiemi di nodi differenti in cardinalità anche qualche ordine di grandezza. Ciò implica che l'ordine con cui i lavori sono completati non rispetta in generale quello con cui essi hanno avuto inizio. La soluzione in esame presenta dunque la grave inefficienza di forzare il supervisore nell'attesa del risultato relativo all'esplorazione della mossa più promettente, anche nell'eventualità che siano già disponibili i risultati di altri lavori. L'organizzazione dei risultati in una struttura di tipo bag elimina invece questo problema permettendo la gestione di un risultato non appena esso viene generato.

5.2.1.1.2 Il flusso di controllo interno dei processi.

A completamento della presentazione dell'algoritmo di base segue l'analisi del flusso di controllo interno dei moduli supervisore ed esploratore.

Si consideri inizialmente il modulo supervisore: la relativa implementazione in Linda è descritta in Fig. 5.4. Il codice proposto separa la fase di gestione centrale della ricerca (funzione master) dalle fasi di inizializzazione e terminazione della stessa (funzione master_main). Tale decomposizione intende isolare la parte caratterizzante dell'algoritmo (la fase centrale di ricerca) dalle altre aventi carattere generale e che non saranno inutilmente replicate nella successiva presentazione di alcune varianti dell'algoritmo.

```
/* le costanti simboliche NEWPOS e QUIT
identificano tipi di lavoro differenti dalla
normale ricerca αβ di alberi di gioco:
rispettivamente la ricezione di un nuovo
albero e la terminazione del processo*/
#define NEW_POSITION -1
#define QUIT -2
int master_main(int n_worker,int depth)
{
```

```
position *root,*successor;
int nmoves,minimax,i,master(),worker();
for (i=0;i<n_worker;i++)
       eval ("worker",worker(depth));  /*
creazione struttura di worker */
while (!end())
       {
       load_new_position (&root); /* un nuovo
albero di gioco */
       nmoves=genmoves(root,&successor);
       if (nmoves==0)   /* non è necessaria
nessuna ricerca? */
              return(evaluate(root));
       out ("root",*root); /* comunica ai
worker il nuovo albero */
       out ("sincr",n_worker);
       for (i=0;i<n_worker;i++)
              out ("job",NEW_POSITION); /*
richiesta di lettura del nuovo albero */
       in ("sincr",0); /* il master prosegue
solo dopo che tutti i worker hanno ricevuto
       il nuovo albero */
       minimax=master(n_worker,nmoves,depth,su
ccessor); /* fase centrale della ricerca */
       in ("root",*root);  /* rimozione
dell'albero visitato */
       }
for (i=0;i<n_worker;i++)
       {
       out ("job",QUIT);  /* comunicazione di
terminazione dell'elaborazione */
       in ("worker", 0);  /* la terminazione
di un worker fa sì che la
       relativa tupla attiva divenga passiva e
       quindi rimovibile dallo spazio delle
tuple*/
       }
return (0);
}
int master(int n_worker,int nmoves)
{
int nmoves,local_score,subtree,free,best;
local_score=-INFINITE;  /* inizializzazione
copia locale dello score */
out ("score", local_score);  /*
inizializzazione score globale */
free = n_worker;
subtree = 1;
while ((subtree <= nmoves) || (free<n_worker))
/* il ciclo termina quando sono stati
       distribuiti tutti i lavori e recuperati
       tutti i risultati */
       if ((subtree <= nmoves) && (free>0)) /*
c'è ancora lavoro da distribuire e almeno
       uno degli esploratori è libero? */
              {
              out ("job",subtree); /*
inserimento in agenda di un lavoro di ricerca
*/
              free--;
              subtree++;
              }
       else     /* attesa del completamento di
almeno un lavoro */
              {
              in ("result",?value,?r_subtree);
/* recupero di un risultato */
              if (value>local_score)  /*
miglioramento dello score? */
                     {
```

```
                        local_score=value;     /*
aggiornamento locale dello score */
                        in ("score",?int);
                        out("score",local_score);
/* aggiornamento globale dello score */
                        best=r_subtree;
                        }
                free++;
                }
in ("score",?int);  /* la tupla "score" va
rimossa per non produrre effetti laterali
       indesiderati nelle successive ricerche
*/
return (local_score);
}
```

Fig. 5.4 Algoritmo base: flusso di controllo del processo master

---

La funzione master_main descrive una successione ciclica di ricerche di alberi di gioco intendendo simulare così la situazione generale che si verifica in una partita reale. Tale sequenza è preceduta da una fase iniziale in cui viene creata la struttura di processi worker. In particolare viene creata una tupla attiva per ciascuno di essi; tale tupla innesca l'esecuzione asincrona della funzione worker (Fig. 5.5). Detta funzione ha un unico parametro formale: la profondità nominale di ricerca; questo parametro di ricerca può essere già comunicato in questa fase perché assunto costante in tutte le ricerche. Normalmente nella lista di parametri formali è presente anche un identificatore (o nome) del generico worker che permette al master di stabilire, in situazioni eccezionali, interazioni di tipo simmetrico con esso. La sua assenza nella funzione worker evidenzia ancora maggiormente la completa indistinguibilità di questi processi.

Creazione e terminazione di processi sono attività computazionalmente costose; non appaiono quindi efficienti soluzioni che prevedono la creazione di una struttura di worker per ogni nuova ricerca. L'approccio corretto è invece che nella ricerca di tutti gli alberi di gioco venga utilizzata la stessa struttura di worker. In questo e molti altri domini è quindi certamente da preferire il progetto di un unico tipo di modulo worker con funzionalità generali, capace però di specializzarsi quando necessario impedendo così di ricorrere ad una sua sostituzione

qualora l’elaborazione richieda una modifica dinamica delle funzioni ad esso richieste.

```
int worker(int depth)
{
position *root,*successor,*tree_pointer;
int
nmoves,subtree,score,value,job_type,quit,s;
quit=false;
while(!quit)
       {
       in ("job",?job_type);
       switch (job_type)
              {
              case QUIT:  /* fine */
                     quit=true;
                     break;
              case NEW_POSITION: /* la ricerca
riguarda nuovo albero di gioco */
                     in ("position",?*root);
                     in ("sincr",?s);
                     out ("sincr",s-1);

       nmoves=genmoves(root,&successor);
                     break;
              default:    /* il job riguarda
una normale ricerca sequenziale αβ */
                     subtree=job_type;

       tree_pointer=successor+subtree-1;
                     makemove(tree_pointer);
                     rd ("score",?score);  /*
aggiornamento score locale */
                     value=-
alphabeta(tree_pointer,-INFINITE,-
score,depth);

       out ("result",value,subtree);
                     undomove(tree_pointer);
                     break;
              }
       }
return (0);
}
```

Fig. 5.5 Algoritmo base: flusso di controllo di un processo worker

Nel caso di problemi di ricerca, qualora esistano parametri che differenzino una ricerca dalla precedente, è ragionevole che la comunicazione di questi (da master a worker) avvenga come una normale interazione mediata da scambio di tuple passive. Nell’algoritmo in esame tale interazione riguarda unicamente la comunicazione per diffusione del nodo radice del nuovo albero di gioco; essa avviene prima dell’inizio della ricerca. Si osservi come il master completi questa interazione sincronizzandosi con la ricezione di questa informazione da parte di

tutti i moduli worker. Punti di sincronizzazione come questo sono normalmente origine di inefficienza; il loro inserimento è però necessario al fine di garantire la correttezza dell'algoritmo. Nel caso specifico dell'implementazione di Fig. 5.4 le velocità relative dei processi potrebbero essere tali per cui, in assenza del punto di sincronizzazione, un modulo worker potrebbe ricevere un lavoro di ricerca prima della specifica dell'albero di gioco cui essa si riferisce.

La ricezione del nuovo nodo radice costituisce per i worker un lavoro di natura diversa rispetto alla normale ricerca $\alpha\beta$; lo stesso vale per la comunicazione di fine elaborazione formulata dal master una volta completato il ciclo di ricerche. In situazioni come questa di lavori con diverso significato come può il processo worker identificare il tipo di lavoro prelevato dall'agenda?

La soluzione generale in Linda è inserire nella tupla-lavoro un campo che ne indichi esplicitamente il tipo. Nella presente implementazione è stata possibile un'ottimizzazione: il tipo di lavoro e la sua descrizione occupano lo stesso campo della tupla-lavoro. I lavori di "lettura nodo radice" e "fine elaborazione" non sono infatti accompagnati da descrizione; prevedendo per essi identificatori numerici diversi dai possibili valori che può assumere la descrizione del lavoro "ricerca $\alpha\beta$", l'identificazione di quest'ultimo tipo di lavoro avviene implicitamente per esclusione.

La fase ciclica della funzione master_main si chiude con la rimozione di tutte le tuple passive ancora presenti nello spazio omonimo. Seppure non necessaria ai fini della ricerca (ormai completata), questa operazione è indispensabile nel garantire la correttezza: la presenza di tuple "spurie" potrebbe infatti creare interazioni indesiderate durante le successive ricerche.

Si consideri la funzione master; essa descrive la fase centrale di distribuzione e controllo della ricerca. Il flusso di controllo evidenzia l'intercalarsi durante la ricerca complessiva

delle due funzionalità predominanti del modulo master:

• creazione ed inserimento in agenda di un nuovo lavoro e

• ricezione di un risultato e conseguente gestione.

L’implementazione proposta dà priorità alla prima attività: viene generato un nuovo lavoro non appena un modulo esploratore si rende disponibile per la sua elaborazione. L’informazione "numero di esploratori disponibili", seppure di natura globale ai moduli, viene gestita localmente dal supervisore; egli, infatti, è in grado di dedurre che un esploratore non è più occupato dal verificarsi dell’evento "ricezione di un risultato". Il modulo supervisore è così in grado di stabilire autonomamente quando inserire in agenda un nuovo lavoro: immediatamente dopo la ricezione di un risultato.

Il codice della funzione master descrive il procedere della ricerca attraverso tre fasi consecutive:

• start-up: è la fase iniziale in cui il master genera un numero di lavori pari a quello degli esploratori; quest’ultimi vengono quindi impegnati nella loro totalità fin dall’inizio della ricerca. Ad epilogo di questa attività il supervisore si sospende nell’attesa della ricezione del primo dei risultati che saranno inseriti nell’omonima struttura dati distribuita.

• fase intermedia: in questo stadio la ricezione di un risultato e la produzione di un nuovo lavoro di ricerca si alternano regolarmente. Questo comportamento del supervisore limita però il tempo di occupazione del generico esploratore il quale, dopo aver prodotto un risultato, è costretto ad attendere che esso sia ricevuto ed elaborato dal supervisore prima di poter disporre di un nuovo lavoro. Una possibile soluzione a questo problema è rilasciare l’eccessivo accoppiamento supervisore-esploratore facendo sì che il numero medio L di lavori in agenda sia non nullo. Da un punto di vista implementativo è sufficiente che nella fase di start-up siano inseriti in agenda ($L+N_w$) lavori, dove N_w è il numero di esploratori. La costante L deve

essere dimensionata in modo da sopportare variazioni improvvise delle velocità relative dei moduli (ad es. esploratori che completano rapidamente la ricerca) o situazioni eccezionali (ad es. completamento simultaneo di più lavori); essa non può però superare certi valori di soglia a causa dei problemi di mancato ordinamento dei lavori ed overhead di ricerca che ne deriverebbero. Questo tipo di soluzione sarà tuttavia sperimentato nello studio dei metodi di decomposizione dinamica. L'inefficienza presentata sarà invece eliminata con un approccio differente, applicato ad una delle varianti dell'algoritmo di base che verranno presentate in seguito (cfr. 5.2.1.1.2).

• attesa finale: è la fase conclusiva in cui il supervisore, dopo avere distribuito la ricerca di tutti i sottoalberi al top-level, attende la ricezione dei risultati degli ultimi lavori non ancora completati. Questa fase rivela una grave inefficienza: l'agenda di lavori rimarrà vuota fino alla fine della ricerca e quindi ogni esploratore diverrà definitivamente inoperoso dopo avere completato il proprio lavoro e ciò fino a che tutti gli altri non avranno terminato il rispettivo. Tale problema costituisce un caso particolare di distribuzione del carico non uniforme. I metodi di decomposizione statica soffrono inevitabilmente di questo problema, a meno che essi prevedano un'analisi statica del carico la quale permetta di programmare il completamento pressoché contemporaneo degli ultimi lavori (essa è comunque di non realistica attuazione per problemi di ricerca $\alpha\beta$). I metodi con decomposizione dinamica, invece, risolvono in modo naturale i problemi della natura appena descritta.

Si consideri, infine, la funzione worker di Fig. 5.5 la quale descrive il flusso di controllo di un generico esploratore. Esso si sviluppa attraverso il ciclico ripetersi delle operazioni:

• estrazione di un lavoro dall'agenda
• identificazione del tipo di lavoro ricevuto e sua esecuzione.

Il codice proposto mette in risalto l'operazione di selezione del tipo di lavoro e separa in modo netto le azioni corrispondenti a ciascuna

classe; il dettaglio di quest'ultime è già stato implicitamente discusso.

5.2.1.1.3 I risultati sperimentali

In questa sezione sono presentati ed analizzati i risultati della sperimentazione dell'algoritmo base applicato al dominio degli scacchi.

Tale algoritmo e tutti i successivi che verranno presentati saranno sottoposti agli stessi test di valutazione delle prestazioni; questa scelta permetterà immediati confronti. Le considerazioni che seguiranno riguardo l'organizzazione degli esperimenti devono quindi intendersi di carattere generale.

L'algoritmo verrà impegnato nella esplorazione di posizioni di test; in particolare è stato scelto l'insieme di 24 posizioni di Bratko-Kopec [BraKop82] (riportato integralmente in Appendice A). Si tratta di posizioni appositamente ideate per valutare la qualità di gioco di un giocatore automatico di scacchi. Deve essere chiaro, tuttavia, che questa sperimentazione intende stimare unicamente l'efficienza dell'algoritmo distribuito e non la qualità delle sue scelte; quest'ultima, infatti, dipende pesantemente dalla conoscenza del dominio confinata nella funzione di valutazione e la cui analisi esula dagli scopi di questa sezione.

La finalità principale degli esperimenti è quella di confrontare le prestazioni dell'algoritmo distribuito con quelle della sua versione sequenziale. L'algoritmo base di decomposizione sarà confrontato ovviamente con l'algoritmo $\alpha\beta$. Alcuni degli algoritmi paralleli presentati in seguito conterranno euristiche per il miglioramento della ricerca: il loro algoritmo sequenziale di riferimento sarà in tal caso alphabeta arricchito delle stesse euristiche.

La sperimentazione degli algoritmi è accompagnata dalla raccolta di parametri statistici con i quali si intende porre in risalto vari aspetti della dinamica della loro esecuzione.

I parametri più importanti per la valutazione di un programma parallelo riguardano i tempi di elaborazione. Essendo l'architettura hardware costituita da una rete di workstation, cioè

macchine multiutente e quindi non completamente dedicate all'esecuzione di un'unica applicazione, classifichiamo i possibili tempi di elaborazione in:

• tempo reale (o assoluto): misura la durata effettiva degli esperimenti, quale cioè apparirebbe ad un osservatore umano che assistesse al loro evolversi;

• tempo di macchina (o di CPU): misura il tempo realmente dedicato dalla macchina agli esperimenti, "depurando" così il tempo reale delle frazioni in cui la macchina serve altri utenti.

Data la visita di una generica posizione di test, il parametro temporale più interessante è il tempo complessivo T di ricerca dell'albero di gioco ad essa associato.

Per l'algoritmo sequenziale il tempo di ricerca è misurato come l'intervallo che intercorre fra l'invocazione al top-level della funzione ricorsiva $\alpha\beta$ ed il suo completamento. Le prestazioni dell'algoritmo sequenziale costituiscono il metro di valutazione degli algoritmi distribuiti e quindi la loro misura non può essere influenzata dalle condizioni di carico (per loro natura variabili) della macchina su cui essa è operata. Per questo motivo il tempo complessivo T viene misurato come tempo di macchina.

Il tempo di ricerca dell'algoritmo parallelo viene misurato dal processo master: esso è l'intervallo fra l'inizio della distribuzione del nuovo nodo radice e l'acquisizione del suo valore minimax finale.

Per gli algoritmi paralleli è molto difficoltoso l'utilizzo del tempo di macchina quale metrica: ciò richiederebbe l'analisi dei tempi macchina di ogni workstation impegnata nella ricerca ed una loro complessa sintesi in unico dato statistico. Il tempo di ricerca verrà dunque misurato come tempo reale; questa scelta non pregiudica la validità degli esperimenti poiché essa comporta un'approssimazione per difetto dell'effettivo valore degli algoritmi paralleli[22].

Il confronto fra l'algoritmo parallelo e la sua versione sequenziale viene sintetizzato dallo speedup (o guadagno) S, inteso come rapporto tra il tempo complessivo T_s di ricerca

sequenziale ed il tempo complessivo T_p impiegato dalla versione parallela sullo stesso albero di gioco:

$$S = \frac{T_S}{T_P}$$

Lo speedup S è un indice assoluto indicante il guadagno in termini di velocità computazionale introdotto dalla sostituzione dell'algoritmo sequenziale con una sua versione parallela.

Altro parametro assoluto è l'efficienza relativa E; esso normalizza lo speedup S rispetto al numero di processori impiegati:

$$E = \frac{S}{N} = \frac{T_S}{T_P \cdot N}$$

L'efficienza relativa è utile quale misura del degrado complessivo delle prestazioni; essa infatti scaturisce dal confronto fra il guadagno reale e quello ideale.

I parametri statistici sin qui definiti sono di tipo generale ed utilizzati come metriche standard nella valutazione dell'efficienza di algoritmi paralleli. La loro definizione intende implicitamente che i processori sono identici. Ciò che in generale avviene quando l'architettura hardware è costituita da una rete locale è che le risorse di elaborazione sono eterogenee e quindi con prestazioni differenti. In questo contesto le prestazioni degli algoritmi sequenziale e parallelo sono difficilmente confrontabili attraverso i parametrici statistici S ed E. Per tenere conto di questa situazione il calcolo del tempo complessivo di ricerca T deve essere corretto di un fattore che dipende dalle velocità relative di elaborazione delle singole macchine:

• sia l'insieme di processori $P=\{P_1,P_2,...,P_N\}$ e $P_r \in P$ un processore di riferimento;

• sia v_i la velocità di elaborazione[23] del processore P_i $(i=1..N)$;

• ad ogni processore è associato un peso di che esprime la velocità relativa di elaborazione rispetto al processore P_r:

$$d_i = \frac{v_i}{v_r} \quad (i = 1..N)$$

• il fattore di correzione f_c è così calcolato:

$$f_c = \frac{N}{\sum_{i=1}^{N} d_i}$$

• Dato il tempo di ricerca complessivo T, il valore che sarà effettivamente assunto per questo parametro è:

$f_c \cdot T$

Il fattore medio di produzione F_{pm} è un parametro che permette di valutare l'effettivo impiego produttivo dei processori durante il tempo complessivo T di ricerca; esso è così definito:

• sia N il numero totale dei processori e T_{p_i} (i=1..N) il tempo dedicato dall'i-esimo processore ad attività produttiva, cioè all'esecuzione di lavoro del tipo "ricerca αβ";

• la media aritmetica T_{pm} dei singoli contributi T_{p_i} (i=1..N) esprime il tempo che in media un processore dedica alla ricerca; il fattore medio di produzione normalizza questo parametro rispetto alla durata T della ricerca costituendo così un indice assoluto:

$$F_{pm} = \frac{T_{pm}}{T} = \frac{\sum_{i=1}^{N} T_{p_i}}{T}$$

Nell'algoritmo di base proposto il modulo master non esegue attività produttiva e quindi il suo tempo di produzione è nullo. In questo caso particolare verrà riportato anche il tempo medio di produzione dei soli moduli worker (F_{pmw}), così da isolare il comportamento eccezionale del master.

Il parametro F_{pm} consentirà di stimare il bilanciamento del carico indotto dall'algoritmo parallelo; è infatti tale proprietà ad influire maggiormente sull'utilizzo produttivo dei processori. Più in generale esso darà indicazioni sull'incidenza complessiva dei degradi legati alla sincronizzazione ed alla comunicazione.

Altro importante dato statistico è la dimensione della ricerca; essa è quantificata dal numero totale N di nodi dell'albero visitati (interni+terminali).

Il confronto fra le dimensioni della ricerca sequenziale (N_s) e parallela (N_p) permette di stimare l'overhead di ricerca OR:

$$OR = \frac{N_p}{N_s} - 1$$

Parametro tipico di un algoritmo di ricerca è la misura VR della quantità di ricerca elaborata nell'unità di tempo (velocità di ricerca):

$$VR = \frac{N}{T}$$

Anche per questo parametro è possibile il confronto fra la sperimentazione nel sequenziale e nel parallelo; a riguardo è calcolato un indice assoluto analogo allo speedup:

• siano VR_s e VR_p rispettivamente le velocità di ricerca sequenziale e parallela;

• si definisce guadagno di velocità S_{vr} il parametro:

$$S_{vr} = \frac{VR_p}{VR_s}$$

Nell'ipotesi realistica di overhead di ricerca non negativo vale la relazione: $S_{vr} \geq S$. Si osservi, infatti, che sul valore dell'indice Svr non incide l'overhead di ricerca; esso consente quindi di isolare e valutare gli effetti delle altre forme di degrado delle prestazioni.

La sperimentazione effettiva dell'algoritmo di base è stata preceduta da una serie di test di verifica delle velocità di elaborazione delle singole workstation. Ciascuna di esse è stata impegnata nella ricerca sequenziale con profondità 5-ply della posizione nr.1 di Bratko-Kopec; quale indice della velocità di elaborazione è stata considerata la velocità di ricerca (espressa in nodi/sec). Questi test hanno evidenziato una limitata eterogeneità delle 11 macchine a disposizione (SUN 4) evidenziando una partizione delle stesse in due gruppi di macchine omogenee (la differenza relativa di prestazioni di workstation appartenenti allo stesso gruppo è inferiore al 5%):

• 8 workstation (compresa quella di riferimento) meno veloci;

• 3 workstation con velocità di elaborazione superiore di un fattore 1.7 alla macchina di riferimento.

Questi dati consentono di calcolare il fattore di correzione del tempo complessivo di ricerca; tale valore è funzione di numero e tipo di

workstation impiegate nella ricerca parallela. Ad esempio nel caso di sperimentazione con tutte le 11 macchine esso è costante:

$$f_c = \frac{\sum_{i=1}^{N} \frac{v_i}{v_r}}{N} = \frac{8 \cdot 1 + 3 \cdot 1.7}{11} \cong 1.19$$

Un parametro della ricerca di alberi di gioco è la sua profondità nominale. Per tutti gli esperimenti essa è stata fissata a 5-ply. Le motivazioni di questa scelta sono duplici: la sperimentazione su profondità inferiori appare non significativa date le limitate dimensioni dell'albero, mentre la scelta di profondità maggiori di 5-ply è resa proibitiva dai tempi eccessivi necessari al suo completamento.

Tutti gli algoritmi paralleli implementati sono parametrici rispetto al numero di processori impiegati nella ricerca. L'esplorazione delle posizioni di Bratko-Kopec verrà ripetuta con diversi valori di questo parametro, così da tracciare la dipendenza funzionale dei parametri statistici rispetto ad esso. Lo studio della loro variabilità permetterà inoltre di anticipare il comportamento dell'algoritmo in presenza di un numero di processori ancora maggiore del massimo a disposizione per l'esecuzione di questi esperimenti.

Per motivi di spazio non sarà riportato il dettaglio dei parametri statistici calcolati per ogni posizione di test, bensì una loro naturale generalizzazione basata sulla somma dei tempi di ricerca e del numero di nodi visitati nel ciclo di esplorazione di tutte le posizioni.

Alcune considerazioni di carattere teorico intendono spiegare quali valori di efficienza si devono attendere dalla sperimentazione di tale algoritmo:

• l'efficienza dell'algoritmo base è proporzionale al rapporto B/N_w fra il fattore di diramazione dell'albero di gioco ed il numero di esploratori [MarCam82];

• per un albero di gioco con valore tipico B=40, una ricerca sequenziale $\alpha\beta$ è equivalente ad una ricerca esaustiva (algoritmo minimax) di un albero con fattore B=7 di diramazione [Gil72]. Pertanto, se l'algoritmo di decomposizione fosse sperimentato su un'architettura di N_w=40

esploratori, lo speedup medio che si dovrebbe ottenere è 7.

In Tab. 5.1b è presentato il risultato della sperimentazione dell'algoritmo base di decomposizione, mentre in Tab. 5.1a è spiegato il significato delle colonne di Tab. 5.1b (tale legenda avrà valore anche per le successive tabelle).

nr. proc	il numero complessivo di processori (master+worker)
T	il tempo complessivo di ricerca (misurato in secondi) necessario alla esplorazione di tutte le 24 posizioni di test
N	il numero totale di nodi visitati nella ricerca di tutte le posizioni di test
S	lo speedup $(=\frac{T_1}{T_{nr.proc}})$
E	l'efficienza relativa $(=\frac{S}{nr.proc})$
Fpm	il fattore medio di produzione
Fpmw	il fattore medio di produzione dei worker
SO	l'overhead di ricerca $(=\frac{N_{nr.proc}}{N_1}-1)$
VR	la velocità di ricerca $(=\frac{N}{T})$
SVR	il guadagno di velocità $(=\frac{VR_{nr.proc}}{VR_1})$

Tab. 5.1a Sommario dei parametri statistici

nr. proc	T(sec)	N (nodi)	S	E	Fpm	Fpmw	SO	VR (nodi/sec)	Svr
1	8301	5347545	---	---	---	---	---	644	---
3	4571	5754203	1.82	0.61	0.66	0.98	0.08	1259	1.95
5	2770	6599460	3.00	0.60	0.75	0.93	0.23	2382	3.70
7	2242	7236560	3.70	0.53	0.71	0.83	0.35	3228	5.01
9	1988	7876769	4.18	0.46	0.68	0.77	0.47	3962	6.15
11	1786	8521288	4.65	0.42	0.66	0.73	0.59	4771	7.41

Tab. 5.1b Algoritmo base di decomposizione: sperimentazione

Il dato statistico più rilevante è rappresentato dai valori molto bassi dello speedup. Il motivo di prestazioni tanto deludenti risiede nella forte incidenza complessiva delle forme di degrado; il parametro E (efficienza relativa) spiega inoltre come essa sia tanto maggiore quanto più elevato è il numero di processori.

L'esame in dettaglio di ciascuna forma di degrado evidenzia una diversa variabilità in funzione del numero di processori:

• overhead di ricerca: l’aumento della quantità di nodi visitati procede di pari passo con quello dei processori; il degrado maggiore è stato infatti ottenuto dalla versione dell’algoritmo con 11 processori che ha visitato il 59% di nodi in più rispetto all’algoritmo sequenziale. L’incidenza di questa forma di degrado sulle prestazioni è notevole: si confrontino a riguardo i valori dello speedup S e i rispettivi guadagni S_{vr} nella velocità della ricerca; quest’ultimo dato statistico può essere interpretato come lo speedup che si sarebbe ottenuto in assenza di overhead di ricerca. I dati statistici relativi alla ricerca con 11 processori, ad esempio, rivelano una perdita di efficienza del 37% causata da questa forma di degrado (S_{vr}=7.41 >> S=4.65).

L’entità del degrado si rivela molto elevata se confrontata con valori tipici di altri algoritmi. La causa di tale comportamento è dovuta alla minima condivisione: i moduli esploratori cooperano (con mediazione del supervisore) solamente all’inizio ed alla fine della propria ricerca (rispettivamente aggiornando la copia dello score e comunicando il risultato della ricerca). Durante la fase centrale è dunque loro negata la possibilità di fruire dei risultati prodotti dagli altri esploratori, inducendo così una ricerca più svantaggiosa rispetto all’algoritmo sequenziale dove la ricerca di un sottoalbero può avvalersi dei risultati di tutte le ricerche precedenti (Fig. 5.6).

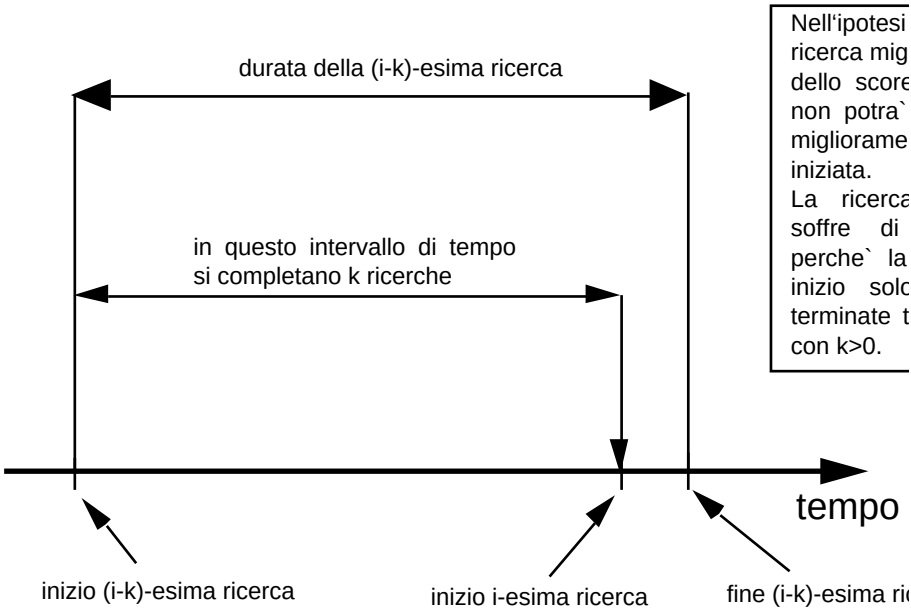


Fig. 5.6 Dipendenza dell'overhead di ricerca dalle velocità relative dei processi

Perché l'overhead di ricerca è funzione crescente del numero di processori?

La ricerca $\alpha\beta$ trae beneficio dal calcolo quanto più immediato di un elevato valore parziale dello score; il suo valore iniziale è $-\infty$ e con esso avranno inizio le visite dei primi Nw sottoalberi della radice (Nw è il numero di esploratori): è evidente il degrado rispetto alla ricerca sequenziale dove già la visita del secondo sottoalbero potrà contare su una buona approssimazione del valore ottimo dello score (quella calcolata dalla ricerca del primo sottoalbero). Il fenomeno non riguarda solo la ricerca dei sottoalberi iniziali: in generale la maggiore distribuzione aumenta anche la probabilità che una ricerca R_i non benefici del miglioramento dello score prodotto dalla ricerca R_{i-k} (attivata precedentemente) proprio perché R_i ha avuto inizio prima che R_{i-k} fosse completata.

• overhead di sincronizzazione e comunicazione: il fattore medio di produzione F_{pm} fornisce indicazioni su queste forme di degrado. Si osservi lo strano andamento di questo fattore in funzione del numero di processori: crescente fino a 5 processori (75%) e poi decrescente fino a 11 dove si ottiene lo stesso fattore acquisito con 3 processori

(66%). Si confrontino questi valori con il fattore di produzione medio riferito ai soli esploratori il quale presenta andamento costantemente decrescente. La causa della discordanza fra la variabilità dei due parametri di produzione è l'attività non produttiva (dal punto di vista della ricerca $\alpha\beta$) del modulo supervisore. Esso svolge sole mansioni di coordinatore e per la maggior parte del tempo complessivo di ricerca T è sospeso in attesa di interagire con gli esploratori: questa situazione è tipica in implementazioni basate sul modello master-worker. Nell'ipotesi (finora assunta implicitamente) che ogni modulo sia allocato su un processore diverso, in virtù dell'elevato tempo di sospensione del supervisore si viene a creare un forte inutilizzo del processore che lo ospita. Soluzione immediata a questo problema è che il supervisore ed uno degli esploratori siano allocati sullo stesso processore. Tuttavia tale soluzione è non descrivibile nello standard di Linda essendo l'allocazione dei processi trasparente al programmatore; si consideri, inoltre, che il paradigma master-worker opera efficientemente solo se il master è in grado di generare lavoro velocemente in modo da tenere occupati tutti i worker: se esso condividesse il tempo di CPU con un altro processo i suoi tempi di risposta potrebbero essere così rallentati da influire sensibilmente sulle prestazioni complessive.

All'aumentare del numero di esploratori la rinuncia alla produttività del processore che ospita il supervisore diviene meno influente; per contro, cresce il numero medio di esploratori inattivi durante la fase finale della ricerca (il fattore F_{pmw} è infatti decrescente). Sono questi i motivi di degrado che affliggono maggiormente la produzione: la loro combinazione giustifica l'andamento non monotono del fattore F_{pm}. Il costo delle comunicazioni incide invece sulla produzione in modo costante e trascurabile: il numero di interazioni fra moduli sono infatti in numero esiguo (proporzionale al fattore di diramazione dell'albero) ed indipendente dalla quantità di processori.

5.2.1.2 Varianti dell’algoritmo base di decomposizione: l’impiego del master nella ricerca

I risultati sperimentali hanno sottolineato i riflessi sull’efficienza del mancato impiego nella ricerca $\alpha\beta$ del processore destinato all’esecuzione del processo supervisore. Non disponendo di strumenti in Linda per il controllo dell’allocazione dei processi, il recupero "produttivo" del tempo in cui il supervisore è sospeso in attesa del completamento di un esploratore deve essere programmato esplicitamente. A riguardo sono presentate due differenti soluzioni.

5.2.1.2.1 Un approccio intuitivo

Si vuole applicare all’algoritmo base di decomposizione l’idea seguente: una volta che è stato distribuito lavoro sufficiente ad impegnare tutti gli esploratori, il modulo supervisore esegue il lavoro relativo al primo sottoalbero della radice non ancora distribuito; solamente dopo aver completato tale compito esso sarà nuovamente assegnato alle sue mansioni di supervisione. Durante la fase di ricerca il supervisore diviene a tutti gli effetti un esploratore: l’architettura logica di comunicazione di Fig. 5.1 deve essere dunque modificata per tenere conto di questa nuova situazione (Fig. 5.7).

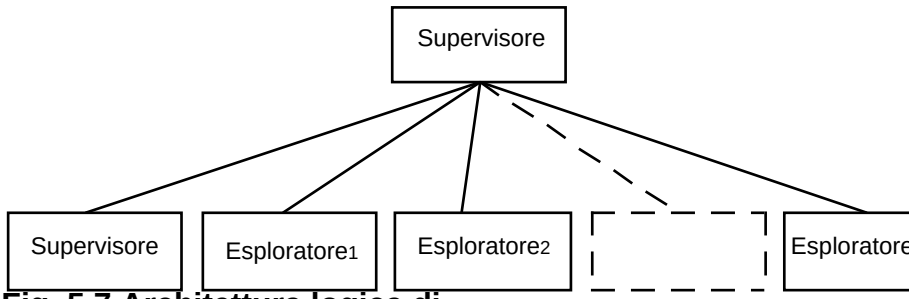


Fig. 5.7 Architettura logica di comunicazione con impiego nella ricerca del processo supervisore

Le modifiche apportate al modulo supervisore sono trasparenti ai moduli esploratori il cui flusso di controllo rimane invariato rispetto alla prima versione dell’algoritmo base. Esse sono inoltre confinate nella fase centrale della

ricerca che per il modulo supervisore è descritta dalla funzione master in Fig. 5.8.

```
#define update_local_and_global_score\
{\
local_score=value;\
in ("score",?int);\
out ("score",local_score);\
best=r_subtree;\
}\
int master(int n_worker,int nmoves,int
depth,position *successor)
{
int nmoves,local_score,subtree,free,best;
local_score=-INFINITE;
out ("score", local_score);
free=n_worker;
subtree=1;
while ((subtree <= nmoves) || (free<n_worker))
       if (subtree<=nmoves)
              {
              if (free>0) /* si distribuisce
lavoro appena un worker è libero*/
                     {
                     out ("job",subtree);
                     free--;
                     subtree++;
                     }
              else
                     {
                     if (inp
("result",?value,?r_subtree)) /* si testa se
un esploratore
       ha completato il suo lavoro */
                            free++;
                     else  /* invece di
sospendersi nell'attesa dei
       risultati degli esploratori il
supervisore
       diviene lui stesso esploratore */
                            {

       tree_pointer=successor+subtree-1;

       makemove(tree_pointer);
                            value=-
alphabeta(tree_pointer,-INFINITE,-
local_score,depth);

       undomove(tree_pointer);
                            r_subtree=subtree;
                            }
                     if (value>local_score)

       update_local_and_global_score;
/* la gestione dei risultati è realizzata
dalla stessa porzione di codice,
indipendentemente dal fatto che essi siano
prodotti dagli esploratori o dal supervisore
*/
                     }
              }
       else  /* attesa finale del
completamento degli ultimi lavori */
              {
              in ("result",?value,?r_subtree);
              if (value>local_score)
```

```
        update_local_and_global_score;
               free++;
               }
in ("score",?int);
return (local_score);
}
```

Fig. 5.8 Impiego nella ricerca del supervisore: un approccio intuitivo

Il codice in Linda evidenzia la priorità che il modulo supervisore assegna alla gestione dei lavori rispetto alla sua nuova attività di esploratore: quest'ultima è infatti intrapresa solo se:

- (free==0): è stato distribuito un numero di lavori sufficiente ad impegnare tutti gli esploratori e
- (inp("result",?value,?r_subtree)==false): nessun esploratore ha completato la sua ricerca

Il modulo supervisore implementa in modo diverso rispetto agli esploratori le fasi iniziale e finale dell'esecuzione di un lavoro: il prelievo di quest'ultimo e la comunicazione del relativo risultato non sono infatti mediati da strutture dati distribuite; lo scambio logico di queste informazioni avviene all'interno dello stesso modulo e non è quindi necessaria alcuna interazione sullo spazio delle tuple.

I dati statistici di Tab. 5.2 descrivono i risultati della sperimentazione di questa nuova versione dell'algoritmo base.

nr. proc	T (sec)	N (nodi)	S	E	Fpm	SO	VR (nodi/sec)	Svr
1	8301	5347545	---	---	---	---	644	---
3	4544	5860668	1.83	0.61	0.70	0.10	1290	2.00
5	3173	6330053	2.62	0.52	0.66	0.18	1995	3.10
7	2779	6867688	2.99	0.43	0.61	0.28	2471	3.84
9	2517	7310806	3.30	0.37	0.57	0.37	2905	4.51
11	2310	7990369	3.59	0.33	0.54	0.49	3459	5.37

Tab. 5.2 Sperimentazione dell'impiego nella ricerca del supervisore

L'approccio proposto si è rivelato fallimentare: il confronto con la prima versione dell'algoritmo base evidenzia una notevole riduzione dello speedup, maggiormente visibile con l'aumentare del numero di processori.

La spiegazione di risultati tanto deludenti è ancora una volta suggerita dall'esame del fattore di produzione Fpm il quale evidenzia un tempo di utilizzo troppo esiguo delle risorse di elaborazione. Con 11 processori, ad esempio, si ha F_{pm}=0.54: ciò significa che un generico processore è stato inoperoso durante il 46% del tempo complessivo di ricerca. Tali valori possono essere giustificati solo da una strategia di distribuzione dei lavori inefficiente. Un modulo esploratore che produce un risultato durante la fase centrale della ricerca diviene inoperoso fino a che il supervisore non rileva questo evento e produce di conseguenza un nuovo lavoro. Nella nuova versione dell'algoritmo il modulo supervisore è impegnato per la maggior parte del suo tempo di elaborazione nella ricerca di sottoalberi e le funzioni di supervisione possono solo intercalarsi fra una ricerca e la successiva; a causa del suo duplice compito il supervisore non è in grado di rilevare prontamente il completamento di un altro lavoro: ciò avverrà solo dopo che avrà terminato il suo lavoro corrente. Pertanto un generico esploratore verrà sospeso per un tempo che in media è pari alla metà del tempo medio di ricerca di un sottoalbero della radice.

Per effetto dell'aumentata inoperosità dei processori si è però avuta una diminuzione dell'overhead di ricerca. I due parametri sono infatti direttamente collegati: il fatto che i moduli esploratore si sospendano più a lungo in attesa dell'attribuzione di un nuovo lavoro aumenta la probabilità che quest'ultimo non inizi prima che anche altri esploratori abbiano completato il proprio e possa così beneficiare dell'eventuale aggiornamento dello score da essi prodotto.

Il miglioramento dell'overhead di ricerca è comunque irrisorio rispetto all'entità del problema legato all'eccessivo inutilizzo dei processori. In conclusione il confronto fra i due algoritmi proposti vede dunque vincente la prima versione evidenziando così come la rinuncia di un processore alla ricerca sia comunque conveniente se ciò comporta un

elevato bilanciamento del carico fra i moduli worker.

5.2.1.2.2 Una modifica del paradigma master-worker: un modello di cooperazione alla pari

Le due versioni dell'algoritmo base sin qui descritte hanno evidenziato prestazioni insoddisfacenti. Le principali cause di tale inefficienza sono rispettivamente:

• l'impiego di un processore (quello su cui è allocato il modulo supervisore) durante la fase centrale di ricerca per sole finalità di gestione

• l'indisponibilità immediata di nuovo lavoro per l'esploratore che ha completato la ricerca di un sottoalbero.

La presenza di questi problemi non è intrinseca nell'algoritmo di base, ma scaturisce dal paradigma di programmazione parallela con il quale è stato implementato: il modello master-worker. La presente sezione intende apportare alcune modifiche a tale modello al fine di limitare (o addirittura eliminare) entrambi i motivi di degrado elencati.

L'idea che si intende sperimentare è confinare il ruolo del master a quello di solo generatore di lavori; tale attività, inoltre, viene completata prima che la ricerca effettiva abbia inizio. La gestione dei risultati prodotti dal lavoro degli esploratori è invece completamente a carico di quest'ultimi.

Immediata conseguenza di questo approccio è che si rende inutile la presenza di un processo master durante la ricerca dell'albero di gioco; la risorsa di elaborazione da esso occupata può essere quindi assegnata all'esecuzione dei lavori senza creare problemi di ritardo nella distribuzione dei lavori. L'attività di gestione della ricerca del modulo supervisore sarà dunque decomposta in tre fasi consecutive:

• generazione ed inserimento in agenda di tutti i lavori;

• conversione nel ruolo di esploratore;

• rilevamento dell'avvenuto completamento di tutti i lavori e prelievo del risultato finale.

Nella prima fase il modulo supervisore inserisce in agenda i lavori relativi alla ricerca di tutti i sottoalberi della radice. Tale approccio

rende insoddisfacente l'implementazionde dell'agenda come una struttura dati distribuita di tipo bag in quanto l'indistinguibilità dei lavori indotta da questa struttura ne sopprime l'ordinamento. Nelle versioni precedenti dell'algoritmo questo effetto poteva essere considerato trascurabile poiché l'esecuzione ordinata dei lavori veniva controllata dal modulo supervisore il quale si limitava a generare solo i lavori necessari ad occupare gli esploratori inoperosi: il "disordine" prodotto dall'agenda avrebbe comunque interessato in generale un numero esiguo di lavori consecutivi (nell'ordine di generazione). L'inserimento contemporaneo (da un punto di vista logico) di tutti i lavori in un'agenda di tipo bag induce, invece, un ordine di prelievo dei lavori completamente casuale. Per questi motivi la struttura di lavori deve essere implementata come una coda; la struttura dati distribuita che soddisfa tale requisito è la coda-in (cfr. 3.4.1.3.2). Alcune proprietà dell'algoritmo base rendono tuttavia sufficiente una versione semplificata di tale struttura dati: si osservi infatti che la descrizione dei lavori (l'indice del sottoalbero della radice da visitare) coincide con la posizione che essi occupano nella sequenza che si intende mantenere ordinata. Pertanto l'indice di estrazione della coda (head) contiene di per se tutta l'informazione necessaria alla descrizione di un lavoro. Di conseguenza l'agenda sarà implementata da un'unica tupla della forma:

```
("head",subtree)
```

La sua presenza indica che è già stata impegnata la ricerca dei primi (subtree-1) sottoalberi della radice e la successiva avrà per oggetto il subtree-esimo.

L'operazione logica di inserimento di tutti i lavori è dunque ridotta alla inizializzazione della tupla appena detta:

```
out ("head",1);
```

Il prelievo di un nuovo lavoro è invece implementato dall'incremento logico del campo subtree:

```
in ("head",?subtree);
out ("head",subtree+1);
```

Sia NMOVES il numero di mosse al top-level: il campo subtree non è più incrementato dopo che ha raggiunto il valore (NMOVES+1), cioè dopo che anche l'ultimo lavoro è stato prelevato. La presenza della tupla ("head", NMOVES+1) permetterà di riconoscere lo stato di terminazione della ricerca.

Completata l'inizializzazione della struttura di lavori il modulo supervisore diviene un esploratore alla pari di tutti gli altri. Scompare dunque la figura di un amministratore e controllore unico della ricerca: la gestione globale non è più centralizzata, ma distribuita fra gli esploratori.

L'elaborazione dei risultati della ricerca compete ora ai moduli esploratore: essi cooperano infatti aggiornando il valore corrente dello score al top-level; questa informazione è condivisa da tutti i moduli in quanto implementata come variabile distribuita. In particolare l'esploratore che completa la ricerca di un sottoalbero verifica lui stesso se il risultato da essa prodotto migliora il valore corrente dello score. La consistenza di queste operazioni viene garantita dalla indivisibilità dei costrutti Linda:

```
in ("score",?old_score);
new_score=(value>old_score) ?
value : old_score;
out ("score",new_score);
```

Fig. 5.9 sottolinea le modifiche apportate all'architettura logica di comunicazione evidenziando la gestione alla pari dello score corrente in opposizione a quella centralizzata del modello master-worker.

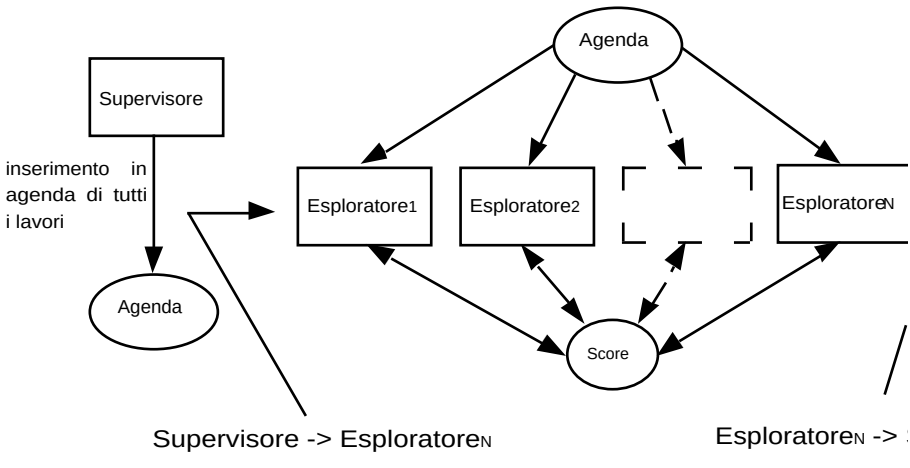


Fig. 5.9 Architettura logica di cooperazione alla pari

L’applicabilità del modello di cooperazione presentato è certamente meno generale di quella del modello master-worker rigoroso. Sono infatti proprietà particolari dell’algoritmo base che rendono questo modello una valida struttura per un’implementazione efficiente:

• è possibile stabilire a priori l’identità dei lavori che dovranno essere eseguiti: in molte applicazioni i nuovi lavori che sono generati dipendono dai risultati prodotti dall’esecuzione dei precedenti e quindi il metodo non è applicabile poiché la generazione dei lavori deve avvenire dinamicamente durante il tempo della loro esecuzione.

• il numero dei lavori realmente necessari è fissato prima dell’inizio della ricerca: spesso nei problemi di ricerca, pur essendo definibile staticamente l’insieme di lavori da completare, i risultati intermedi rendono un sottoinsieme di essi non necessario. Questa sarebbe stata la situazione se la finestra $\alpha\beta$ al top-level non fosse stata $(-\infty,+\infty)$, ma (α,β) e lo score parziale avesse superato il limite β: avendo ottenuto un taglio la ricerca dei restanti sottoalberi è inutile e tale è quindi anche l’inserimento iniziale in agenda dei relativi lavori. In generale l’inefficienza introdotta è proporzionale al numero medio di lavori "tagliati" ed alle dimensioni della loro descrizione.

• l’elaborazione dei risultati dei lavori richiede un tempo trascurabile rispetto alla loro esecuzione. In generale questa operazione agisce su informazioni globali ai moduli e deve essere eseguita in modo indivisibile: se la sua durata non fosse trascurabile essa sarebbe origine di frequenti collisioni per l’aggiornamento delle strutture condivise, di conseguenti serializzazioni degli accessi alle relative sezioni critiche e in definitiva di un aumento dell’overhead di sincronizzazione. La minimizzazione della durata delle sezioni critiche viene proposta in letteratura come uno dei requisiti fondamentali per la limitazione del degrado della performance.

• il generico worker è in grado di rilevare in modo autonomo la condizione terminale di "agenda vuota". Questa condizione è necessaria affinché il modulo supervisore, "confuso" durante la ricerca fra gli altri esploratori, sia in grado di riconoscere la fine dei lavori per potere ripristinare il suo ruolo di privilegio.

Nella terza e conclusiva fase della ricerca, infatti, il supervisore riacquista la sua singolarità. In particolare esso si sincronizza con l'evento "completamento di tutti i lavori": il fatto che il supervisore abbia già rilevato l'assenza di lavori in agenda non implica infatti che gli ultimi lavori prelevati siano stati tutti completati. Quest'ultimo evento viene riconosciuto indirettamente grazie al rilevamento di un altro evento di cui esso è logica conseguenza: "tutti gli esploratori hanno rilevato la condizione di agenda vuota". Solo dopo tale verifica il valore corrente dello score può essere assunto come finale e letto dalla relativa variabile distribuita.

Fig. 5.10 descrive in Linda le idee sin qui descritte presentando l'algoritmo base così modificato alle prese con la fase centrale della ricerca.

La funzione worker_search è eseguita sia dal modulo supervisore che dagli esploratori; per quest'ultimi essa è invocata all'interno di una funzione worker più esterna (non presentata in figura) dopo che è avvenuta la ricezione del nodo radice del nuovo albero di gioco da esplorare. La coincidenza della funzione di controllo interno della ricerca sottolinea il ruolo completamente alla pari (con i moduli esploratore) che il modulo supervisore assume durante questa fase.

```
int master(int n_worker,int nmoves,int
depth,position *successor)
{
int nmoves,score,best;
void worker_search();
out("score",-INFINITE); /* inizializzazione
score globale */
out("ending_workers",n_worker+1);
out("head",1);   /* riempimento agenda ≈
inizializzazione puntatore alla testa della
coda-lavori */
```

```
worker_search(nmoves,depth,successor); /*
supervisore >> esploratore */
in("ending_workers",0); /* attesa
completamento di tutti i lavori ≈ tutti gli
esploratori hanno
rilevato l'evento "agenda vuota" */
in("head",nmoves+1);
in("score",?score); /* prelievo score finale
*/
return(score);
}
void worker_search(int nmoves,int
depth,position *successor)
{
int
subtree,tree_pointer,value,initial_score,curre
nt_score,quit,e_w;
quit=false;
while (!quit)
       {
       in("head",?subtree); /* prelievo lavoro
≈ rimozione puntatore di estrazione */
       if (subtree==(nmoves+1)) /* agenda
vuota? */
              {
              out("head",subtree); /* la
condizione di terminazione deve poter essere
rilevata
       anche dagli altri esploratori */
              in ("ending_workers",?e_w);
              out ("ending_workers",e_w-1);
              quit=true;
              }
       else
              {
              out("head",subtree+1); /*
aggiornamento puntatore di estrazione */
              tree_pointer=successor+subtree-1;
              makemove(tree_pointer);
              rd ("score",?initial_score); /*
aggiorna copia locale dello score */
              value=-alphabeta(tree_pointer,-
INFINITE,-initial_score,depth);
              undomove(tree_pointer);
              if (value>initial_score) /*
migliorata la copia locale dello score? */
                     {
                     in
("score",current_score); /* inizio sezione
critica di aggiornamento
       dello score globale */
                     if (value>current_score)
/* miglioramento? */

       current_score=value;
                     out
("score",current_score); /* fine sezione
critica */
                     }
              }
       }
}
```

Fig. 5.10 Implementazione dell'algoritmo base con cooperazione alla pari

La sperimentazione dell'algoritmo presentato, documentata in Tab. 5.3, intende verificare il

miglioramento dei motivi di degrado che hanno afflitto le prime due versioni dell'algoritmo base.

nr. proc	T (sec)	N (nodi)	S	E	Fpm	SO	VR (nodi/sec)	Svr
1	8301	5347545	---	---	---	---	644	---
3	3298	6156027	2.52	0.84	0.96	0.15	1867	2.90
5	2382	6945194	3.48	0.70	0.91	0.30	2916	4.53
7	2064	7524137	4.02	0.57	0.81	0.41	3645	5.66
9	1798	8186074	4.62	0.51	0.77	0.53	4553	7.07
11	1751	8746752	4.74	0.43	0.71	0.64	4995	7.75

Tab. 5.3 Algoritmo base con cooperazione alla pari: sperimentazione.

In Fig. 5.11 è riportato il confronto fra le curve dello speedup della nuova versione dell'algoritmo base e della prima proposta. Il miglioramento delle prestazioni può essere osservato su tutte le configurazioni di processori; il motivo di questo risultato è dovuto alla maggiore produttività media dei processori. Come atteso, infatti, gli esploratori riescono a cooperare in modo efficiente nella ricezione dei lavori e nell'elaborazione dei relativi risultati.

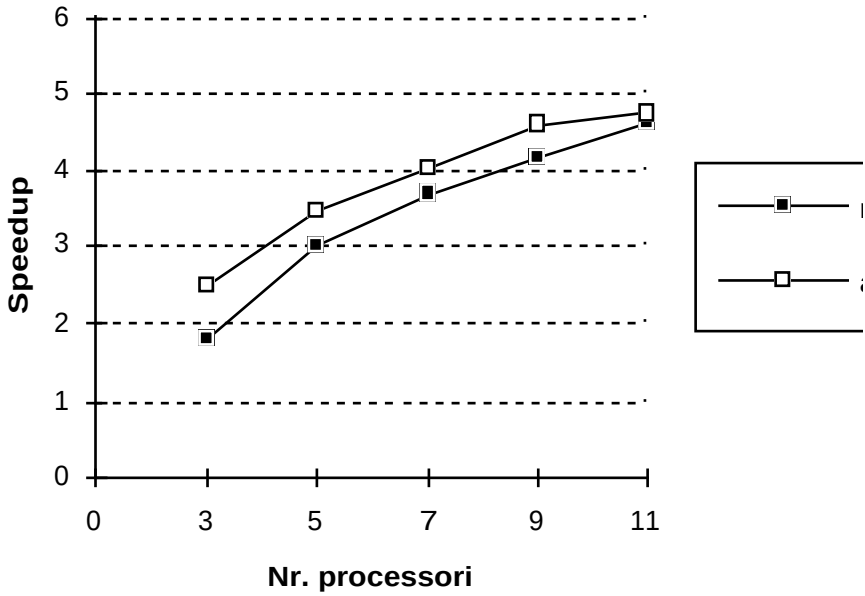


Fig. 5.11 Confronto dello speedup delle versioni dell'algoritmo base con cooperazione master-worker e alla pari

Un dato sorprendente è costituito dal sensibile aumento dell'overhead di ricerca: esso ha una duplice spiegazione:

• rispetto alla versione basata sul modello master-worker rigoroso il parallelismo reale nella ricerca è aumentato di una unità: ciò implica una maggiore distribuzione (decomposizione) dell'albero di gioco e quindi un aumento dell'overhead di ricerca;

• il tempo di attesa del generico esploratore fra il completamento di un lavoro (nel presente caso si intende esecuzione ed elaborazione del risultato) e l'inizio del successivo si può assumere minore rispetto alla prima versione in quanto l'assegnamento del lavoro è gestito autonomamente ed è inoltre eliminato il ritardo dovuto al suo inserimento in agenda. Conseguenza della riduzione di questo tempo di attesa è una minore probabilità per un'esploratore di usufruire dell'aggiornamento prodotto da un altro che ha completato la sua ricerca dopo di esso.

Si osservi come la differenza fra le prestazioni dei due algoritmi a confronto diminuisca con l'aumentare del numero di processori fino quasi ad annullarsi in presenza di 11 di essi. La situazione descritta non è casuale: con l'aumento dei processori, infatti, i due algoritmi tendono a ridurre i vantaggi e svantaggi reciproci fino ad una situazione limite in cui essi hanno le stesse prestazioni. Tale intuizione è confortata da alcune considerazioni di carattere analitico:

• sia P il numero complessivo di processori e sia E(P) il numero di processori impiegati come esploratori;

• sia il coefficiente C(P) così definito:

$$C(P) = \frac{E_{master\ -\ worker}(P)}{E_{alla\_\ pari}(P)} = \frac{P-1}{P}$$

• il rapporto fra i fattori di produzione dei due algoritmi è proporzionale al coefficiente C(P), mentre il rapporto fra i rispettivi overhead di ricerca è inversamente proporzionale ad esso;

• si osservi che $\lim_{p\to\infty} C(P) = 1$ e $\lim_{p\to\infty} \frac{1}{C(P)} = 1$

• in conclusione, assumendo (con realistica approssimazione) che le principali forme di degrado siano funzione del solo numero di

processori E(P) impegnati nella esplorazione, si può affermare che i due algoritmi si equivalgono asintoticamente nelle prestazioni.

5.2.1.3 Lo studio degli algoritmi su alberi di gioco ordinati: approfondimento iterativo e riordinamento dei nodi al top-level

Tutti gli algoritmi finora presentati prevedono lo sviluppo di un'unica ricerca condotta fino alla profondità nominale dell'albero di gioco. L'ordinamento dei nodi al top-level in questo caso è povero in quanto fissato dal generatore di mosse. L'efficienza della ricerca $\alpha\beta$ è fortemente condizionata dall'ordine di visita dei nodi (soprattutto di quelli al top-level); fondare la sperimentazione degli algoritmi $\alpha\beta$ paralleli sulla ricerca di alberi di gioco il cui grado di ordinamento è impredicibile può quindi condurre a grossolani errori di valutazione.

Si è così scelto di uniformare la ricerca a soli alberi di gioco ordinati: di conseguenza essa deve essere arricchita di strumenti per l'ordinamento dinamico dei nodi.

In particolare la ricerca unica a profondità nominale è sostituita da una successione di ricerche definite in rispetto dell'euristica denominata approfondimento iterativo. La versione "distribuita" di questa euristica è perfettamente analoga a quella sequenziale:

- è fissata una profondità iniziale di ricerca INIT ed una massima MAX (pari alla profondità nominale);
- viene condotta fino a profondità INIT la ricerca parallela dell'albero di gioco (ad esempio guidata dall'algoritmo base di decomposizione);
- la ricerca in parallelo viene reiterata a profondità aumentata di una unità. Questo procedimento viene ripetuto fino a raggiungere profondità MAX di ricerca.

Si osservi che le dimensioni degli alberi di gioco esplorati nelle prime iterazioni potrebbero essere così ridotte da rendere non conveniente un approccio parallelo alla loro ricerca: la decomposizione di questi alberi potrebbe infatti causare un costo delle comunicazioni molto più elevato di quello di una ricerca sequenziale. La soluzione generale è dunque fissare una profondità di soglia al di sopra della quale la ricerca è parallela, mentre per profondità inferiori essa è realizzata sequenzialmente. Questo accorgimento determina un'evidente overhead di sincronizzazione: se P è il numero di processori, (P-1) di essi sono inoperosi per tutta la durata delle prime

ricerche sequenziali. Tale degrado può comunque considerarsi trascurabile poiché questo tempo di attesa è irrisorio rispetto a quello necessario a completare la ricerca a profondità superiori.

L'euristica di approfondimento iterativo è finalizzata a creare uno scenario di ricerca che favorisca l'efficacia della quasi totalità delle euristiche di ordinamento dei nodi; per il momento, tuttavia, verrà considerata la sola euristica denominata riordinamento dei nodi al top-level.

L'idea alla base di questa euristica è portare in testa all'ordinamento dei nodi al top-level quelli la cui valutazione ha prodotto, durante la i-esima iterazione, un miglioramento dello score; gli effetti di questo riordinamento si rifletteranno positivamente sulla (i+1)-esima e successive iterazioni.

L'applicazione di quest'ultima euristica all'algoritmo base con cooperazione alla pari comporta l'introduzione di nuove strutture dati distribuite:

• l'ordine dei nodi al top-level deve essere noto a tutti i moduli; non è più possibile distribuire questa informazione implicitamente poiché l'ordinamento non è quello fissato dal generatore di mosse, ma varia da un'iterazione all'altra. Supponendo un numero NMOVES di mosse al top-level si renderà necessaria la memorizzazione del loro ordine in una struttura dati distribuita di tipo vettore di NMOVES elementi ciascuno individuato dalla tupla con schema:

```
("order",index,subtree)
```

il cui significato è: il sottoalbero subtree occupa la posizione index nell'ordinamento corrente.

• il prelievo di un lavoro si articola ora in due fasi:

• lettura della posizione che esso occupa nella sequenza:

```
in ("head",?INDEX);
out ("head",INDEX+1);
```

• lettura dal vettore di ordinamento della descrizione del lavoro corrispondente all'indice letto:

```
in ("order",INDEX,?subtree);
```

Si osservi come l'insieme di tuple e operazioni su esse indicate definiscano nel complesso una struttura dati di tipo coda-in.

• le informazioni necessarie all'aggiornamento del vettore di ordinamento sono prodotte dagli esploratori durante la ricerca; esse si compongono di:

• una variabile distribuita con funzione di contatore del numero di miglioramenti dello score ottenuti durante l'iterazione corrente:

```
("counter",count)
```

• una lista ordinata distribuita contenente l'identificatore dei sottoalberi la cui valutazione ha provocato miglioramento dello score:

```
("improver",index,subtree)
```

Tali strutture dati sono così aggiornate: all'interno della sezione critica che definisce per il modulo esploratore l'aggiornamento dello score globale, se si verifica un miglioramento del suo valore, vengono eseguite le seguenti operazioni:

```
in ("counter",?count);
count=count++;
out ("counter",count);
out ("improver",count,subtree);
/* subtree = indice del sottoalbero
visitato */
```

• l'aggiornamento del vettore di ordinamento è eseguito dal modulo master; esso avviene prima che abbia inizio una nuova iterazione (da un punto di vista logico questa operazione coincide con l'inserimento ordinato in agenda di tutti i lavori). In particolare viene scandita secondo l'ordine di inserzione la lista distribuita di sottoalberi che hanno migliorato lo score: ciascuno di essi viene portato in testa all'ordinamento complessivo delle mosse al top-level. Questo procedimento fa sì che il primo sottoalbero presente nel vettore di ordinamento sia quello relativo all'ultimo aggiornamento dello score o in altre parole risultato migliore durante l'iterazione precedente.

La sperimentazione dell'algoritmo così ottenuto è stata effettuata fissando i parametri dell'euristica di approfondimento iterativo come segue: INIT=2-ply, MAX=5-ply e la soglia per l'applicazione della ricerca parallela pari a 3-ply.

Quale algoritmo sequenziale di confronto è stato considerato $\alpha\beta$ arricchito delle versioni sequenziali delle due euristiche descritte (le profondità INIT e MAX sono le stesse scelte per la versione parallela).

nr. proc	T (sec)	N (nodi)	S	E	Fpm	SO	VR (nodi/sec)	Svr
1	6740	4529971	---	---	---	---	672	---
3	2884	5415125	2.34	0.78	0.94	0.20	1878	2.79
5	2229	6307062	3.02	0.60	0.87	0.39	2830	4.21
7	2007	7317370	3.36	0.48	0.80	0.62	3646	5.42
9	1839	8187769	3.67	0.41	0.75	0.81	4452	6.62
11	1825	8980505	3.69	0.34	0.71	0.98	4921	7.32

Tab. 5.4 Algoritmo base con approfondimento iterativo e riordinamento dei nodi al top-level: sperimentazione

Si confrontino i dati sperimentali di Tab. 5.4 relativi alla nuova versione dell'algoritmo base con quelli ottenuti in assenza delle nuove euristiche di ordinamento dei nodi (Tab. 5.3). Il dato statistico più allarmante è il notevole aumento dell'overhead di ricerca (98% con 11 processori!!). Si osservi come il numero assoluto N dei nodi visitati sia quasi invariato ed è addirittura maggiore in presenza di 9 e 11 processori. La versione sequenziale, al contrario, trae grosso beneficio dall'inserimento delle nuove euristiche: la porzione di albero visitata è ridotta del 17%.

I motivi di un contributo così diverso delle euristiche alla ricerca parallela rispetto alla sequenziale è ancora dovuto al problema della ricerca dei primi P sottoalberi (P è il numero dei processori): per essi l'ordinamento reciproco non ha valore perché sono rimossi contemporaneamente al fine di impegnare da subito tutti i processi; la ricerca di questi sottoalberi si presume sia molto più lunga rispetto ai successivi a causa dello score iniziale per loro ancora pari a $-\infty$: di conseguenza le tecniche di ordinamento dei nodi apportano vantaggi minimi (per l'algoritmo base).

Si deve considerare, inoltre, che rispetto all'algoritmo con assenza di approfondimento iterativo, il conteggio dei nodi tiene conto anche dei nodi visitati durante tutte le iterazioni che precedono l'ultima: questo overhead giustifica l'aumento in assoluto del numero di nodi registrato con numero di processori superiore a 9.

Si osservi, infine, una diminuzione in generale del 2% del fattore medio di produzione: su questo dato grava l'overhead supplementare di sincronizzazione accumulato durante le ricerche precedenti quella a profondità massima.

5.2.1.4 La riduzione dell'overhead di ricerca

La sperimentazione di alcune versioni dell'algoritmo base di decomposizione ha permesso di evidenziare la notevole riduzione delle prestazioni di cui è causa l'overhead di ricerca. L'analisi di tali versioni ha già evidenziato i principali motivi che originano un degrado così elevato:

• assenza di condivisione dello score durante la ricerca e

• score iniziale pari a -∞ per i primi N_w lavori (N_w = numero di esploratori).

Verranno presentati di seguito due algoritmi che intendono risolvere i problemi appena elencati.

5.2.1.4.1 La condivisione dello score

Si consideri come riferimento l'implementazione dell'algoritmo base che ha evidenziato la migliore performance, cioè quella basata sul modello di cooperazione alla pari. Si vuole applicare a questa struttura la seguente idea: i moduli esploratori aggiornano la rispettiva copia locale dello score anche durante la ricerca di un sottoalbero. Lo scopo di questo nuovo approccio è garantire una maggiore cooperazione fra i moduli esploratore i quali sono così in grado di beneficiare già durante l'esecuzione del lavoro corrente dei miglioramenti prodotti dagli altri.

La struttura in Linda delle versioni dell'algoritmo base già proposte facilita l'applicazione di questa idea in quanto era già prevista la presenza di una variabile distribuita destinata a contenere il valore dello score e identificata dallo schema di tupla:

```
("score",value)
```

L'aggiornamento della copia locale dello score può quindi essere implementata inserendo l'operazione di lettura di tale variabile in qualsiasi punto della ricerca $\alpha\beta$:

```
rd ("score",?global_score)
```

Resta aperta la discussione riguardo la frequenza di lettura: un aggiornamento molto frequente tenderebbe ad annullare l'overhead di ricerca; tuttavia la lettura di una variabile distribuita comporta un'azione sullo spazio delle tuple il cui costo dipende dall'implementazione delle primitive Linda. Tale costo non può considerarsi trascurabile nel caso di approccio con "in distribuita" (quale è il caso del sistema Linda usato nel presente lavoro) in quanto l'operazione rd determina una o più comunicazioni fisiche fra i processori; come conseguenza la frequenza di lettura dello score globale deve essere limitata in modo da impedire che i vantaggi ottenuti con la riduzione dell'overhead di ricerca siano vanificati dall'eccessivo aumento dell'overhead di comunicazione.

La condivisione dello score avrebbe certamente avuto implementazione più efficiente in un sistema Linda con "out distribuita": infatti le caratteristiche del problema sono tali da considerare trascurabile il numero di scritture nello spazio delle tuple (una per lavoro) rispetto al numero di letture di tipo rd (che si vorrebbe idealmente pari al numero di nodi visitati).

È interessante osservare come in questa circostanza il progetto implementativo dell'algoritmo non possa prescindere da considerazioni che riguardano l'implementazione di uno dei livelli sottostanti nella gerarchia di un sistema informativo: il linguaggio di programmazione.

Al fine di rispettare il requisito di efficienza è stato necessario sviluppare una soluzione dipendente dall'implementazione delle primitive Linda.

L'idea alla base dell'approccio adottato è limitare le letture dello score globale alle sole "utili". Si definisce utile una lettura che modifica (e quindi migliora) la copia dello score locale al modulo: se fra due letture consecutive il valore globale dello score non è stato modificato la seconda di queste è da considerarsi (a posteriori) di nessun giovamento alla ricerca e quindi il suo costo computazionale (elevato in un sistema con "in distribuita") decisamente male investito.

L'utilità della sua lettura costituisce per lo score un attributo la cui conoscenza è però globale ai moduli: infatti essi non sono in grado di stabilire autonomamente se il valore dello score è stato modificato dopo la loro ultima lettura. Di conseguenza stabilire l'utilità della lettura comporta in ogni caso un'interazione con gli altri moduli: tuttavia le proprietà dell'implementazione con "in distribuita" permettono di implementare questa operazione con un costo trascurabile.

L'idea è la seguente:

• dopo ogni lettura dello score globale il generico modulo esploratore inserisce nello spazio delle tuple una tupla privata con struttura:

```
("improved",my_identifier)
```

Per tupla privata si intende una tupla dove uno dei campi contiene il "nome" del modulo che la ha creata (tipicamente un identificatore numerico).

• quando un modulo modifica il valore dello score esso rimuove le tuple private di tutti gli altri esploratori;

• un modulo esploratore è così in grado di conoscere l'avvenuto o meno aggiornamento dello score verificando se la sua tupla privata è ancora presente nello spazio delle tuple:

"utilità della lettura" `= rdp ("improved",my_identifier)`

In un sistema con "in distribuita" un modulo ha accesso privilegiato alle tuple private poiché fisicamente già presenti nella propria area locale di memoria. Le operazioni di rimozione di tutte le tuple private ed il tentativo di lettura di una tupla inesistente (rdp senza successo) sono molto costose; tuttavia esse si verificano in numero molto limitato poiché tale è il numero medio di miglioramenti dello score dai quali traggono origine. La situazione che si verificherà con maggiore frequenza è invece la lettura con successo della tupla privata il cui completamento ha costo trascurabile ed il cui significato è "lettura inutile dello score globale".

La descrizione dell'algoritmo è completata dall'indicazione delle fasi della ricerca $\alpha\beta$ in cui viene eseguito il test descritto: nella versione che verrà sperimentata esso avviene prima di valutare ciascuna delle mosse al primo livello del sottoalbero attribuito al generico esploratore.

L'implementazione descritta non rispetta pienamente il principio di indistinguibilità dei processi worker poiché introduce il concetto di nomi di moduli: essi sono stabiliti dal master al momento della creazione dei worker; a ciascuno di essi è comunicato il proprio identificatore. Non è necessario che un modulo conosca il nome degli altri: la rimozione delle tuple private può essere infatti implementata con un ciclo del tipo:

```
while (inp ("improved",?int));
```

Tab. 5.5 illustra la valutazione sperimentale della soluzione proposta.

nr. proc	T (sec)	N (nodi)	S	E	Fpm	SO	VR (nodi/sec)	Svr
1	6740	4529971	---	---	---	---	672	---
3	2689	4995937	2.51	0.84	0.94	0.10	1858	2.76
5	1854	5346599	3.64	0.73	0.88	0.18	2884	4.29
7	1485	5634915	4.54	0.65	0.81	0.24	3795	5.65
9	1315	5936185	5.13	0.57	0.77	0.31	4514	6.72
11	1280	6726213	5.27	0.48	0.74	0.48	5255	7.82

Tab. 5.5 Condivisione dello score: sperimentazione

I risultati ottenuti, se confrontati con quelli relativi alla versione dell'algoritmo in assenza di condivisione (Tab. 5.4), rivelano una sorprendente efficacia del metodo presentato: l'overhead di ricerca SO è ridotto in generale della metà (del 62% nel confronto con 9 processori). La diminuzione di questa forma di degrado si è riflessa positivamente sull'efficienza: lo speedup con il massimo numero di processori è aumentato del 42% (3.69 $\rightarrow$ 5.27).

I risultati per il fattore di produzione F_{pm} sono un'approssimazione superiore del valore reale: essi non sono infatti depurati dei tempi di "improduttività" legati all'aggiornamento della copia locale dello score. L'incidenza del degrado appena citato può tuttavia essere dedotta come segue:

• sia P il numero di processori, Fpmr il fattore di produzione reale e OC una misura del degrado dovuto alla condivisione dello score; sono allora verificate le seguenti relazioni:

$$S_{VR} = P \cdot F_{pmr} \Rightarrow F_{pmr} = \frac{S_{VR}}{P}$$

$$F_{pm} = F_{pmr} + OC \Rightarrow OC = F_{pm} - F_{pmr}$$

• si consideri i valori di questi parametri riguardo la sperimentazione con 11 processori:

$$F_{pmr}(11) = \frac{S_{VR}}{P} = \frac{7.82}{11} = 0.71$$

$$OC(11) = F_{pm} - F_{pmr} = 0.74 - 0.71 = 0.03$$

Il costo delle comunicazioni introdotte dal metodo descritto non è quindi trascurabile; tuttavia gli enormi benefici da esso apportati riguardo la riduzione dell'overhead di ricerca rendono tale costo più che accettabile.

5.2.1.4.2 PVSplit

Fra gli algoritmi classici di decomposizione statica dell'albero di gioco PVSplit (e le sue

varianti) è senza dubbio fra i più efficienti. Esso rappresenta una naturale generalizzazione dell'algoritmo base: la decomposizione di base è applicata ricorsivamente lungo la variante principale così da impegnare inizialmente tutti i processi nella valutazione del primo sottoalbero della radice. Completata questa visita iniziale, la ricerca dei restanti sottoalberi viene sviluppata analogamente a quanto avveniva nella decomposizione base; la differenza fondamentale è che tutti i sottoalberi successivi al primo sono esplorati disponendo di un buono score iniziale: il valore minimax della prima mossa valutata. Tale score sarà tanto migliore quanto più accurato è l'ordinamento iniziale al top-level: l'algoritmo trae quindi indubbio beneficio dall'approccio con approfondimento iterativo in combinazione con il riordinamento dinamico delle mosse al top-level.

Visto come evoluzione dell'algoritmo base, l'obbiettivo che intende perseguire l'algoritmo PVSplit è dunque limitare il gravissimo overhead di ricerca provocato dalla ricerca con score iniziale $-\infty$ del primo gruppo di sottoalberi.

Le proprietà di questo algoritmo sono già state ampiamente discusse in sede di rassegna; questa sezione del lavoro intende descrivere una possibile implementazione in Linda. Il progetto di questa assume come fondamenta l'algoritmo base con cooperazione alla pari. La versione di PVSplit che si vuole realizzare intende infatti applicare tale algoritmo lungo la variante principale dell'albero di gioco. I nodi che compongono tale variante costituiscono dunque i nodi di decomposizione: la distribuzione della ricerca deve avvenire in corrispondenza di ciascuno di essi (Fig. 5.12).

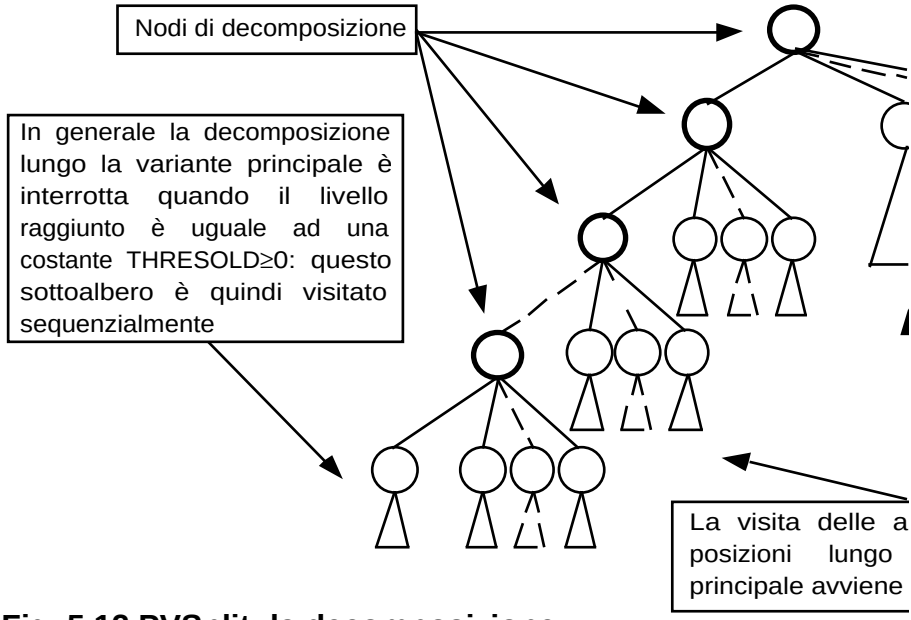


Fig. 5.12 PVSplit: la decomposizione dell'albero di gioco

Si consideri l'i-esimo nodo N_i lungo la variante principale e i suoi N_{MOVES_i} successori:

• la valutazione del primo di essi è eseguita attraverso l'invocazione ricorsiva dello stesso algoritmo PVSplit; la definizione classica dell'algoritmo stabilisce quale condizione di terminazione dell'annidamento ricorsivo il raggiungimento del nodo della variante principale a profondità massima: la sua valutazione è operata dalla funzione di valutazione statica.

Argomenti legati al controllo dell'overhead di comunicazione suggeriscono tuttavia di attivare la decomposizione lungo la variante principale solo fino ad una profondità di soglia, oltre la quale risulta invece più efficiente una visita sequenziale del sottoalbero rimasto.

• la visita dei restanti (N_{MOVES_i}-1) successori è realizzata in rispetto dell'algoritmo base con cooperazione alla pari. Sono possibili anche in questo contesto tutte le ottimizzazioni caratteristiche dell'algoritmo base; condizione essenziale per la loro applicabilità è che tutti i moduli siano a conoscenza del nodo di decomposizione (nella fattispecie N_i e più in generale la porzione della variante principale interessata dalla decomposizione). Nel caso di

generatore di mosse deterministico essi sono in grado di individuare autonomamente la variante principale; nell'ipotesi di ricerca con approfondimento iterativo, tuttavia, la variante principale viene aggiornata dopo ogni iterazione: in questa ipotesi essa deve essere forzatamente memorizzata in una struttura dati distribuita.

```
int master(int n_worker,int depth,int
THRESOLD)
/* il parametro THRESOLD rappresenta il valore
di soglia per il livello di ricerca oltre il
quale la ricerca è sequenziale αβ */
{
position *root;
int nmoves,minimax,i,pvsplit_master(),worker()
;
for (i=0;i<n_worker;i++)
       eval ("worker",worker(depth,THRESOLD));
while (!end())
       {
       load_new_position (&root);
       out ("root",*root);
       out ("sincr",n_worker);
       for (i=0;i<n_worker;i++)
              out ("job",NEW_POSITION);
       in ("sincr",0);
       /* inizio discesa ricorsiva lungo la
variante principale */
       minimax=pvsplit_master(n_worker,root,de
pth,THRESOLD,1);
       in ("root",*root);
       }
for (i=0;i<n_worker;i++)
       {
       out ("job",QUIT);
       in ("worker", 0);
       }
return (0);
}
int pvsplit_master(int n_worker,position
*pv_node,int depth,int THRESOLD,int ply)
{
position *successor;
int nmoves,score;
void worker_search ();
if (ply>THRESOLD)   /* ricerca sequenziale? */
       return (alphabeta (pv_node,-
INFINITE,INFINITE,depth));
nmoves=genmoves(pv_node,&successor);
if (nmoves==0)     /* terminazione della
variante principale per assenza di mosse? */
       return(evaluate(pv_node));
makemove(successor);
/* prosegue la discesa ricorsiva lungo la
variante principale */
score=-
pvsplit_master(n_worker,successor,depth-
1,THRESOLD,ply+1);
undomove(successor);
/* inizio della fase di decomposizione con
algoritmo base delle mosse alternative alla
variante principale */
```

```
out ("score",score); /* lo score iniziale è
ora pari al valore minimax della
       variante principale */
out ("head",2);  /* la decomposizione inizia a
partire dal secondo sottoalbero */
out ("ending_workers",n_workers+1);
/* il supervisore diviene esploratore per
attuare la cooperazione alla pari */
worker_search(nmoves,depth,successor);
in ("head",nmoves+1);
in ("score",?score);
/* score rappresenta il valore minimax per il
nodo lungo la variante principale posto al
livello superiore rispetto a quello in cui è
avvenuta la decomposizione */
return (score);
}
```

Fig. 5.13a PVSplit: flusso di controllo del master

```
int worker(int depth,int THRESOLD)
{
position *root;
int job_type,quit;
quit=false;
while(!quit)
       {
       in ("job",?job_type);
       switch (job_type)
              {
              case QUIT:  /* fine */
                     quit=true;
                     break;
              case NEW_POSITION: /* la ricerca
riguarda nuovo albero di gioco */
                     in ("position",?*root);
                     in ("sincr",?s);
                     out ("sincr",s-1);
                     /* inizio discesa
ricorsiva lungo la variante principale */

       pvsplit_worker(root,depth,THRESOLD,1);
                     break;
                     }
       }
return (0);
}
void pvsplit_worker(position *pv_node,int
THRESOLD,int ply)
{
position *successor;
int nmoves;
void worker_search();
nmoves=genmoves(pv_node,&successor);
if (nmoves>0)
       {
       if (ply>THRESOLD) /* la discesa
ricorsiva prosegue, come per il master, fino
       alla profondità di soglia */
              {
              makemove(successor);
              pvsplit_worker(successor,depth-
1,THRESOLD,ply+1);
              undomove(successor);
              }
       /* il worker è sincronizzato con gli
altri moduli per dare inizio alla
       decomposizione con cooperazione alla
pari */
       worker_search(nmoves,depth,successor);
```

```
        }
return ();
}
```

Fig. 5.13b PVSplit: flusso di controllo di un worker

In Fig. 5.13 è illustrata una descrizione in Linda dell'algoritmo PVSplit basata sulle idee appena proposte.

Le funzioni master e pvsplit_master di Fig. 5.13a descrivono complessivamente il flusso di controllo del modulo supervisore. In particolare la funzione master riguarda le fasi iniziale e finale della ricerca la cui gestione è simile a quella già descritta per l'algoritmo base.

Componente caratterizzante dell'algoritmo è invece la fase centrale di ricerca descritta dalla funzione ricorsiva master_pvsplit. Si osservi la presenza del parametro di soglia THRESOLD ad indicare il livello fino al quale la variante principale può essere valutata in parallelo. La valutazione di tale variante viene fatta risalire dal basso verso l'alto: la prima decomposizione avrà quale nodo radice il nodo al livello THRESOLD della variante principale; la successiva, invece, quello al livello (THRESOLD-1) e così via fino al nodo radice dell'intero albero di gioco.

Durante la risalita verso tale nodo ad ogni livello il modulo supervisore attiva una decomposizione con cooperazione alla pari: la funzione worker_search da esso invocata è la stessa presentata durante l'introduzione di questo metodo (Fig. 5.10). L'architettura logica di comunicazione si trasforma dinamicamente da modello master-worker a modello alla pari (e viceversa) tante volte quanti sono i nodi di decomposizione: il modulo supervisore, infatti, divenuto esploratore per cooperare alla decomposizione dell'i-esimo livello della variante principale, ritorna dopo il completamento di questa al suo ruolo singolare di supervisore per "preparare" la successiva decomposizione al livello (i-1).

La funzione ricorsiva pvsplit_worker in Fig. 5.13b descrive la fase di ricerca di un modulo esploratore. Essa è attivata all'interno della funzione worker non appena è ricevuto il nodo

radice del nuovo albero di gioco. I moduli esploratore "percorrono" la variante principale in sincronia con il supervisore:

• inizialmente essi scendono lungo tale variante fino al nodo al livello THRESOLD: qui attendono che il supervisore completi la valutazione sequenziale dei nodi della variante posti a profondità inferiori e che dia vita alla prima decomposizione inserendo lavoro in agenda;

• il modello di decomposizione alla pari permette ad ogni esploratore di riconoscere l'avvenuto esaurimento dei lavori in agenda e quindi di risalire di un livello lungo la variante principale per poi sincronizzarsi con il modulo supervisore per l'inizio della successiva decomposizione. Questo processo è reiterato fino a raggiungere il nodo radice dove ha luogo la decomposizione finale.

Occorre sottolineare fin d'ora la presenza di punti di sincronizzazione lungo tutta la variante principale esplorata in parallelo dovuti all'attesa del completamento di tutti i lavori relativi alla decomposizione corrente: quale incidenza essi hanno sull'efficienza?

L'algoritmo PVSplit è stato proposto per le sue capacità di ridurre l'overhead di ricerca: quanto vale quantitativamente questo miglioramento?

La sperimentazione in Linda intende risolvere questi quesiti. Quello realmente sperimentato è PVSplit arricchito delle euristiche di approfondimento iterativo (con parametri INIT=2-ply e MAX=5-ply) e riordinamento al top-level; è stato fissato inoltre THRESOLD=3-ply. Infine, gli esperimenti sono stati eseguiti con e senza il metodo di condivisione dello score presentato nella precedente sezione (comunque applicato alla sola decomposizione al top-level).

I risultati di entrambi gli algoritmi PVSplit così ottenuti sono riportati rispettivamente in Tab. 5.6a e Tab. 5.6b.

nr. proc	T (sec)	N (nodi)	S	E	Fpm	SO	VR (nodi/sec)	Svr
1	6740	4529971	---	---	---	---	672	---
3	2693	4949981	2.50	0.83	0.91	0.09	1838	2.73
5	1931	5349308	3.49	0.70	0.82	0.18	2770	4.12

7	1659	5657469	4.06	0.58	0.75	0.25	3410	5.07
9	1502	6130143	4.49	0.50	0.67	0.35	4081	6.07
11	1446	6445828	4.66	0.42	0.60	0.42	4458	6.63

Tab. 5.6a PVSplit: sperimentazione

nr. proc	T (sec)	N (nodi)	S	E	Fpm	SO	VR (nodi/sec)	Svr
1	6740	4529971	---	---	---	---	672	---
3	2640	4770665	2.55	0.85	0.91	0.05	1807	2.69
5	1797	5020905	3.75	0.75	0.84	0.11	2794	4.16
7	1443	5228542	4.67	0.67	0.77	0.15	3623	5.39
9	1255	5358664	5.37	0.60	0.71	0.18	4270	6.35
11	1210	5704328	5.57	0.51	0.64	0.26	4714	7.01

Tab. 5.6b PVSplit con condivisione dello score: sperimentazione

L'obbiettivo prefissato di ridurre l'overhead di ricerca è stato pienamente raggiunto da PVSplit: si osservi come l'entità della riduzione di questa forma di degrado sia ancora maggiore di quella ottenuta dal già sorprendente metodo di condivisione dello score (Tab. 5.5).

Tuttavia l'efficienza complessiva rivelata da PVSplit è inferiore rispetto ad esso; il motivo di questo risultato è spiegato dal confronto fra i fattori di produzione dei due algoritmi: PVSplit presenta valori notevolmente ridotti per questo parametro (del 14% con 11 processori!). La causa di questo crollo della produttività media dei moduli è legata alla già discussa presenza dei punti di sincronizzazione lungo la variante principale. Questo degrado può essere ridotto prevedendo una più accorta distribuzione del carico, tale da impegnare tutti i moduli fino al completamento di ciascuna decomposizione.

A questo overhead deve aggiungersi il costo delle comunicazioni necessarie al completamento di ciascuna decomposizione; ognuna di queste prevede lo stesso numero di comunicazioni, indipendentemente dal livello in cui essa avviene (è una proprietà dell'algoritmo base). Di conseguenza l'overhead di comunicazione è amplificato, rispetto all'algoritmo con sola decomposizione, del numero di nodi della variante principale cui è applicata la valutazione in parallelo.

L'algoritmo che scaturisce dalla combinazione di PVSplit con la tecnica di condivisione dello

score è caratterizzato da un'ancora più sensibile riduzione dell'overhead di ricerca (Tab. 5.6b): la sovrapposizione dei due metodi non riduce, quindi, l'efficacia di entrambi. Questo dato conferma che essi operano il rispettivo miglioramento su porzioni disgiunte dell'albero di gioco:

- PVSplit nella ricerca dei sottoalberi dal secondo al P-esimo (P è il numero di processi)
- il metodo di condivisione nella ricerca dei sottoalberi successivi al P-esimo.

L'efficienza ottenuta da PVSplit in combinazione con la condivisione dello score è la massima fra tutti gli algoritmi paralleli sperimentati. Si osservi che su essa continua comunque a gravare pesantemente l'overhead causato dai punti di sincronizzazione lungo la variante principale: questo problema sarà risolto integrando l'algoritmo con un approccio dinamico alla decomposizione dell'albero di gioco mirato a ridurre lo sbilanciamento nella distribuzione dei carichi di lavoro.

5.2.2 Algoritmi di decomposizione dinamica

Il problema principale che affligge gli algoritmi di decomposizione statica dell'albero di gioco è la difficoltà di ottenere un bilanciamento ottimale del carico. In un'implementazione basata sul modello master-worker essa è motivata dalla proprietà di indivisibilità dei lavori: una volta assegnata ad un processo la ricerca di un sottoalbero essa deve essere completata sequenzialmente. Al momento dell'attribuzione del lavoro è difficile stimarne il costo computazionale necessario al suo completamento. Se questa informazione fosse disponibile, il lavoro potrebbe essere schedulato in base ad un criterio di minimizzazione dei tempi di disoccupazione dei processi. Poiché questa non è la realtà per la classe dei problemi di ricerca $\alpha\beta$, gli algoritmi con decomposizione statica non sono in grado di rimediare dinamicamente a distribuzioni pessime del carico: l'ultimo lavoro ancora in sospeso in un punto di sincronizzazione potrebbe richiedere un tempo arbitrariamente lungo per il suo completamento.

Una soluzione al problema è rendere divisibile il lavoro assegnato ad un esploratore, così da autorizzare quest'ultimo alla decomposizione del sottoalbero di cui è responsabile. Nelle prossime sezioni questa nuova idea verrà applicata all'algoritmo base (e quindi di riflesso a PVSplit) e nel progetto di un algoritmo di decomposizione dinamica più generale.

5.2.2.1 Un primo approccio: algoritmo base con subappalto dinamico della ricerca

L’algoritmo base costituisce, in virtù della sua semplicità, una struttura ideale per la sperimentazione di un approccio dinamico alla distribuzione dell’albero di gioco. In particolare l’attenzione sarà focalizzata sulla sua fase finale; essa ha inizio nell’istante in cui è prelevato l’ultimo lavoro in agenda e si protrae fino al completamento di tutti i lavori. In questo stadio dell’esecuzione dell’algoritmo i moduli che completano la rispettiva ricerca divengono definitivamente inoperosi. L’overhead causato da questa proprietà dell’algoritmo può essere ridotto impegnando tali moduli in aiuto di quelli la cui ricerca è ancora in corso. L’applicazione di questa idea esige che un modulo esploratore sia in grado di raccogliere una o più richieste di collaborazione e di decomporre la ricerca corrente assumendone così il ruolo di supervisore.

Per il momento questa idea sarà applicata alla sola valutazione dei successori della radice; nulla impedisce, tuttavia, che essa possa essere generalizzata anche ai livelli inferiori dell’albero: il fatto che i vantaggi di questo metodo siano intuitivamente maggiori quando applicato ai livelli superiori fa preferire lo studio della versione semplificata di questa tecnica.

Per effetto delle modifiche descritte, identità e numero dei nodi di decomposizione non sono prevedibili a priori; la probabilità che un successore della radice diventi un nodo di decomposizione è proporzionale alla durata della sua valutazione e alla posizione che occupa nell’ordinamento al top-level: il fatto che questo evento si verifichi è però impredicibile in quanto dipendente dalle velocità relative dei moduli. L’algoritmo che scaturisce da questo approccio deve essere dunque classificato come di decomposizione dinamica.

L’assistenza ai moduli che non hanno completato il lavoro al top-level può essere attuata attraverso molte forme di cooperazione; quella che verrà descritta di seguito è quindi solo una possibile soluzione.

Si faccia riferimento all’implementazione dell’algoritmo base con cooperazione alla pari e si consideri il seguente scenario a tempo di esecuzione:

• il modulo esploratore E_f ha completato il proprio lavoro di ricerca al top-level e rileva che l’agenda Atop

di lavori al top-level è vuota: la ricerca complessiva è quindi entrata nella sua fase finale.

E_f si accorge, inoltre, che altri moduli non hanno completato il rispettivo lavoro al top-level e deve quindi comunicare la propria disponibilità a collaborare nella loro esecuzione: questa informazione viene tuttavia notificata ad uno solo fra quelli di essi che ancora non hanno avuto coscienza del nuovo stato della ricerca complessiva (sia E_r tale modulo);

• in conseguenza dell'evento descritto il modulo E_r, cui è affidata la ricerca del sottoalbero S, decompone la porzione di S non ancora esplorata potendo ora disporre dell'aiuto di altri esploratori;

• il modulo E_f non viene impegnato solamente nell'assistenza di E_r, ma di tutti i moduli che hanno tramutato la loro ricerca da sequenziale a parallela.

Fase finale della ricerca: l'agenda è vuota

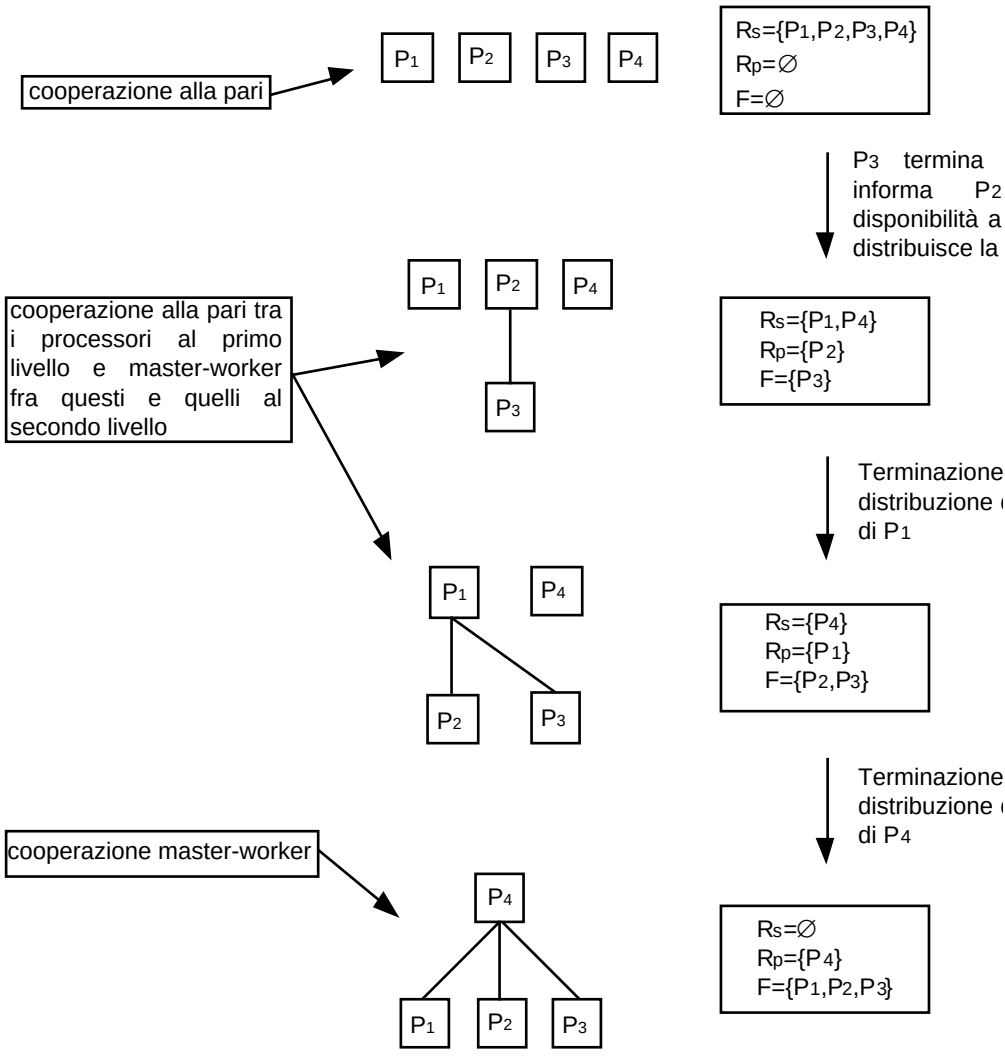


Fig. 5.14 Variazione dinamica dell’architettura logica di comunicazione

L’insieme dei processi è quindi partizionato in tre sottoinsiemi variabili dinamicamente:

• l’insieme F dei processi che hanno completato il lavoro al top-level e offrono la loro collaborazione ai processi "ritardatari";

• l’insieme R_p dei processi che non hanno terminato il proprio lavoro al top-level, ma avendo ricevuto la disponibilità dei moduli in F, ne hanno distribuito l’esecuzione;

• l’insieme R_s dei moduli ritardatari che proseguono nel sequenziale l’adempimento del lavoro di cui sono responsabili.

Quello che accade è dunque che quando un modulo in (R_p U R_s) completa il lavoro al top-level esso diviene membro dell’insieme F e determina il passaggio nell’insieme R_p di un elemento di R_s (a meno che già Rs=∅). La sequenza di eventi descritta in Fig. 5.14 esemplifica quanto appena descritto evidenziando la variazione dinamica dell’architettura logica di comunicazione.

Questo approccio intende limitare il numero di decomposizioni attivate dinamicamente in modo che esso sia al più pari al numero di moduli in F: la decomposizione determina, come noto, costi talvolta elevati e deve quindi essere avviata solo se esiste un sufficiente numero medio di esploratori a suo sostegno.

Il fatto che un modulo in F collabori con tutti i moduli in R_p semplifica l’implementazione della cooperazione che viene originata dal subappalto dei lavori al top-level. È prevista la presenza di un’unica struttura dati distribuita A_{sub} in cui sono inseriti i lavori generati da tutti i moduli (master) in R_p; i moduli (worker) in F eseguono i lavori prelevati da A_{sub} e ne depongono i risultati in una struttura distribuita R_{sub}.

Da un punto di vista logico le strutture A_{sub} e R_{sub} sono in realtà composte da un numero di sottostrutture pari al numero di moduli master. Si può pensare che ad ogni master siano associate una struttura di lavori ed una di risultati; un generico lavoro identifica infatti il modulo che lo ha generato contenendo l’indice ordinale del sottoalbero al top-level di cui esso è responsabile: il modulo worker è così in grado di riconoscere il suo "datore di lavoro" e stabilire in quale delle sottostrutture di R_{sub}

depositare il risultato del lavoro. In particolare la struttura di tupla che descrive un generico lavoro è

```
("job_help",top-subtree,subtree,alpha,beta)
```

Il campo top-subtree è l'identificatore del master appena spiegato; il campo subtree, invece, identifica la mossa di risposta (a quella al top-level con indice top-subtree) che deve essere valutata dal worker. Essendo la decomposizione in analisi attivata ad un livello inferiore a quello della radice non è possibile per il processo worker ricavare autonomamente la finestra $\alpha\beta$ iniziale e deve quindi essere memorizzata fisicamente nella descrizione del lavoro.

Il generico worker vede la struttura A_{sub} come unica in quanto i lavori sono dal suo punto di vista indistinguibili: l'identità del successivo modulo master di cui esso sarà alle dipendenze è stabilita in modo nondeterministico al momento dell'estrazione del lavoro. In questo istante è fissata una relazione gerarchica temporanea con il modulo master che ha generato il lavoro la quale ha durata fino al deposito del relativo risultato.

Lo schema di tupla per un generico risultato è:

```
("result_help",value,top-subtree)
```

A differenza di quanto avveniva nella prima versione dell'algoritmo base, in questo caso la presenza del campo top-subtree è necessaria poiché deve identificare il supervisore cui è indirizzato il risultato (nella circostanza, infatti, esso non è unico).

Si è preferito implementare la cooperazione fra i moduli in R_p e quelli in F secondo il modello master-worker con la variante dell'auto-attribuzione di lavoro da parte del master (cfr. 5.2.1.2.1). Questa scelta è motivata dai seguenti argomenti:

• il problema dovuto all'attesa dei worker che il master termini la ricerca che si è assegnato e possa quindi generare nuovo lavoro è risolto prevedendo che il numero medio di lavori nell'agenda A_{sub} sia non nullo. Il numero di lavori contenuti in essa è un'informazione distribuita fra i moduli: tale valore deve infatti rimanere al di sotto di un certo limite massimo e deve quindi poter essere controllato dinamicamente dai moduli.

Il motivo della limitazione delle dimensioni dell'agenda è di ridurre il tempo medio di deposito di un lavoro: ciò mantiene entro un'approssimazione accettabile l'ordinamento dei lavori e impedisce che il master debba sopportare una lunga attesa per il completamento degli ultimi di essi. Un valore

ragionevole per il limite in questione è il numero corrente di worker, cioè di moduli presenti nell'insieme F. La cardinalità di tale insieme aumenta dinamicamente: di conseguenza questa informazione, essendo globale ai moduli, deve essere anch'essa memorizzata in una struttura dati distribuita. La gestione della struttura di lavori richiede pertanto l'utilizzo di due variabili distribuite con schema:

```
("njobs",nj)
```
e
```
("maxjobs",max)
```
• il metodo di cooperazione alla pari appare non efficace in quanto non può usufruire delle ottimizzazioni possibili nell'algoritmo base: la decomposizione in esame ha luogo al secondo livello dell'albero e può dunque essere terminata da un taglio il che renderebbe inutile l'inserimento in A_{sub} dei lavori relativi alla valutazione dei sottoalberi ancora inesplorati; quest'ultima operazione è costosa poiché, come visto, la descrizione del lavoro è ora informazione più complessa e quindi determina un maggiore overhead di comunicazione.

L'algoritmo presenta dunque un approccio non uniforme all'implementazione della distribuzione della ricerca: mentre la decomposizione statica al livello della radice è implementata attraverso il metodo di cooperazione alla pari, quella dinamica operata al livello dei suoi successori avviene nel rispetto del modello master-worker.

```
#define END_HELP -1
int my_identifier; /* identificatore del modulo */
int master (int n_worker,int nmoves,int
depth,position *successor)
{
int score;
void around_search ();
out ("score",-INFINITE);
out ("head",1);
out ("ending_top-level",n_worker+1);
out ("sincr",n_worker);
out ("njobs",0);
out ("maxjobs",0);
/* supervisore >> esploratore: ha inizio la
decomposizione al top-level con cooperazione alla
pari */
around_search (int nmoves,int depth,position
*successor);
in ("njobs",0);
in ("maxjobs",n_worker);
/* la tupla ("job_help",END_HELP,0,0,0)) indica ai
moduli dell'insieme F che tutti i lavori al top-level
sono stati completati: essa può essere rimossa
soltanto quando è stata letta da tutti essi; la tupla
con etichetta "sincr" permette di controllare il
verificarsi di questo evento */
```

```
in ("sincr",0);
in ("job_help",END_HELP,0,0,0);
in ("score",?score);
return (score);
}
void around_search (int nmoves,int depth,position
*successor)
{
int w,max;
void worker_help();
/* viene inserita nello spazio delle tuple una tupla
privata la cui presenza indica che per il momento non
è possibile usufruire della collaborazione di altri
moduli */
out ("help",my_identifier);
/* la funzione worker_search è la stessa vista
nell'implementazione dell'algoritmo base con
cooperazione alla pari, con la sola eccezione che la
funzione di ricerca sequenziale alphabeta è
sostituita dalla funzione help_alphabeta il cui
codice è presentato in seguito */
worker_search (nmoves,depth,successor);
in ("ending_top-level",?w);
w--;
if (w>0)  /* vi sono ancora moduli che non hanno
completato il lavoro al top-level? */
       {
       in ("maxjobs",?max);
       out ("maxjobs",max+1); /* è aggiornata la
dimensione massima dell'agenda Asub */
       out ("ending_top-level",w);
       inp ("help",my_identifier); /* il modulo ha
completato il lavoro al top-level */
       inp ("help",?int); /* un nuovo modulo può
decomporre dinamicamente la sua ricerca
       corrente */
       /* il modulo inizia il ciclo di prelievo ed
esecuzione dei lavori richiesti dai
       moduli in Rp */
       worker_help (depth,successor);
       }
else  /* il presente modulo ha completato l'ultimo
lavoro al top-level */
       out ("job_help",END_HELP,0,0,0); /* e comunica
questo evento a tutti gli altri
       moduli */
}
void worker_help (int depth,position *successor)
{
int value,w,quit;
position *tree_pointer;
quit=false;
while (!quit)
       {
       in
("job_help",?top-subtree,?subtree,?alpha,?beta);
       if (top-subtree==END_HELP) /* fine
collaborazione? */
              {
              out ("job_help",END_HELP,0,0,0);
              in ("sincr",?w);
              out ("sincr",w-1);
              quit=true;
              }
       else  /* è richiesta l'esecuzione di un nuovo
lavoro */
              {
              in ("njobs",?nj);
              out ("njobs",nj-1);
```

```
                /* si scende di due livelli a partire
dalla radice per individuare il
                sottoalbero di cui è richiesta la
valutazione */
                go_down
(successor,top-subtree,subtree,&tree_pointer);
                /* ora si ha la normale ricerca
sequenziale αβ*/
                value=-
alphabeta(tree_pointer,alpha,beta,depth-1);
                /* vengono "ritirate" le due mosse
top-subtree e subtree ripristinando
                così lo stato iniziale */
                go_up (successor,top-subtree,subtree);
                out ("result",top-subtree,value);
                }
        }
}
int top_alphabeta (position *p,int alpha,int beta,int
depth,int top-subtree)
{
if (depth==0)
        return (evaluate(p));
nmoves=generate(p,&successor);
if (nmoves==0)
        return (evaluate(p));
parallel=false;
wait_result=0;
best=alpha;
for (ind=1;ind<=nmove;ind++)
        {
        tree_pointer=successor+ind-1;
        makemove(tree_pointer);
        dec=false;
/* si continua a verificare la presenza della tupla
privata con etichetta "help" fino a che la sua
assenza comunica al modulo la collaborazione di altri
moduli e quindi l'abilitazione a decomporre la
presente ricerca */
        if (parallel ||
(parallel=!rdp("help",my_identifier)))
                {
                in ("njobs",?nj);
                rd ("maxjobs",?max);
                if (dec=(nj<max)) /* agenda piena? */
                        out ("njobs",nj+1);
                else
                        out ("njobs",nj);
                }
        if (dec) /* deve essere generato un nuovo
lavoro */
                {
                out ("job_help",top-subtree,ind,-beta,-
best);
                wait_result++;
                }
        else /* la valutazione della corrente mossa
avviene in sequenziale */
                value=-alphabeta(tree_pointer,-beta,-
best,depth-1);
        undomove(tree_pointer);
/* viene prima elaborato, se presente, il risultato
della ricerca sequenziale e di seguito viene testata
la presenza dei risultati dei lavori distribuiti in
precedenza (cui segue la relativa elaborazione */
        while ((!dec) || (wait_result && inp
("result",top-subtree,?value)))
                {
                if (!dec)
                        dec=true;
```

```
                else
                        wait_result--;
                if (value>best)
                        best=value;
                if (best>=beta) /* un taglio? */
                        {
/* devono essere rimossi, in quanto inutili, i lavori
inseriti in agenda e "fortunatamente" non ancora
prelevati dai worker */
                        if (parallel && wait_result)
                                remove (top-subtree,
wait_result);
                        return (best);
                        }
                }
/* se la ricerca è stata decomposta dinamicamente
viene attivata l'attesa finale dei risultati dei
lavori non ancora ricevuti */
if (parallel)
        while (wait_result>0)
                {
                in ("result",top-subtree,?value);
                wait_result--;
                if (value>best)
                        best=value;
                if (best>=beta)
                        {
                        if (wait_result)
                                remove
(top-subtree,wait_result);
                        return (best);
                        }
                }
return (best);
}
```

Fig. 5.15 Introduzione di decomposizione dinamica nell'algoritmo base

---

In Fig. 5.15 sono descritte in Linda le modifiche apportate all'algoritmo base per implementare la decomposizione dinamica descritta. Molte delle nuove idee sono state realizzate utilizzando tecniche di programmazione Linda già descritte: si osservi, ad esempio, l'impiego di una tupla privata per minimizzare il costo del test di verifica della disponibilità a collaborare da parte di altri moduli.

La funzione worker_search (Fig. 5.10) che descrive la cooperazione alla pari che ha luogo al livello della radice è ora "avvolta" dalla funzione around_search invocata sia dal modulo master che dai moduli worker (il codice di quest'ultimi non è riportato perché invariato rispetto alla precedente implementazione dell'algoritmo base). Lo scopo di questa funzione è gestire il passaggio di un modulo al ruolo di collaboratore di quegli esploratori che ancora non hanno ancora completato l'esecuzione del proprio lavoro al top-level.

La conversione dinamica della ricerca da sequenziale a parallela, che avviene localmente alla valutazione di uno dei successori della radice, è descritta dalla funzione help_alphabeta; i particolari di criteri e modalità della distribuzione della ricerca verranno chiariti dal progetto di un algoritmo di decomposizione dinamica più generale che verrà proposto nella sezione seguente. Per il momento è interessante osservare che la struttura di base della funzione è l'algoritmo classico alphabeta il cui flusso di controllo viene percorso fedelmente fino a che, rilevata la disponibilità a cooperare di altri moduli, la porzione di albero inesplorato è decomposta sulla base del modello master-worker.

Anche la sperimentazione di questo algoritmo è stata effettuata in condizioni di ricerca con approfondimento iterativo e riordinamento dei nodi al top-level (i parametri caratteristici di queste euristiche sono invariati rispetto agli esperimenti precedenti). I dati statistici ottenuti sono riportati in Tab. 5.7.

nr. proc	T (sec)	N (nodi)	S	E	Fpm	SO	VR (nodi/sec)	Svr
1	6740	4529971	---	---	---	---	672	---
3	3019	5251659	2.23	0.74	0.93	0.16	1740	2.59
5	2216	6322002	3.04	0.61	0.92	0.40	2853	4.24
7	1921	7291629	3.51	0.50	0.91	0.61	3796	5.65
9	1721	8300115	3.92	0.44	0.91	0.83	4823	7.18
11	1600	9439299	4.21	0.38	0.90	1.08	5900	8.78

Tab. 5.7 Algoritmo base e decomposizione dinamica della ricerca: sperimentazione

Si confrontino tali risultati con quelli ottenuti dall'algoritmo base in assenza di decomposizione dinamica (Tab. 5.4). Il dato statistico più importante è l'atteso miglioramento del fattore di produzione medio (0.71→0.90 con 11 processori), sintomo evidente di una più accorta distribuzione del carico. Meno prevedibile era l'attestarsi di tale parametro su valori pressoché costanti ed indipendenti dal numero di processori: la differenza fra i fattori di produzione ottenuti con 3 e 11 processori è appena del 3%. Valori così elevati per questo parametro offrono poco margine di miglioramento alla versione dell'algoritmo in cui la decomposizione dinamica è applicata anche ai livelli inferiori dell'albero di gioco; si può quindi prevedere che i vantaggi della eventuale generalizzazione possano anche essere

completamente vanificati dal costo della sua applicazione.

Per effetto della maggiore decomposizione della ricerca si ha ovviamente un aumento dell'overhead di ricerca il quale raggiunge valori "disastrosi" (108% con 11 processori!). L'aumento di questa forma di degrado tuttavia non annulla il guadagno in efficienza garantito dalla riduzione già descritta dell'overhead di sincronizzazione.

La nuova versione dell'algoritmo base è stata impiegata in PVSplit per implementare ciascuna delle decomposizioni statiche lungo la variante principale. L'algoritmo risultante non è altri che una versione di DPVSplit, la variante di PVSplit presentata in 4.2.5.6.

In Tab. 5.8 è riportata la valutazione sperimentale dell'algoritmo così ottenuto. Esso è inoltre arricchito delle euristiche di approfondimento iterativo e riordinamento al top-level; non è invece inserita la tecnica di condivisione dello score al fine di meglio isolare e valutare i vantaggi prodotti dal nuovo approccio.

nr. proc	T (sec)	N (nodi)	S	E	Fpm	SO	VR (nodi/sec)	Svr
1	6740	4529971	---	---	---	---	672	---
3	2532	4899016	2.66	0.89	0.90	0.08	1935	2.88
5	1805	5453978	3.73	0.75	0.88	0.20	3022	4.50
7	1486	5886412	4.54	0.65	0.87	0.30	3961	5.89
9	1315	6386703	5.13	0.57	0.84	0.41	4857	7.23
11	1208	6789922	5.58	0.51	0.80	0.50	5621	8.36

Tab. 5.8 PVSplit e decomposizione dinamica della ricerca: sperimentazione

Come accaduto per l'algoritmo base, l'aumento del fattore di produzione medio rispetto alla versione classica di PVSplit è tanto più evidente quanto maggiore è il numero di processori; al contrario, il conseguente incremento dell'overhead di ricerca è molto contenuto. Per effetto di queste proprietà della ricerca il miglioramento dal punto di vista dell'efficienza complessiva è notevole: ad esempio lo speedup S con 11 processori ha subito un incremento del 24% passando da 4.50 a 5.58 (cfr. Tab. 5.6a).

5.2.2.2 Un algoritmo generale di decomposizione dinamica

L'algoritmo presentato nella precedente sezione costituisce un approccio molto particolare al concetto di decomposizione dinamica: è possibile individuare staticamente un ristretto insieme dei soli nodi che

possono divenire, dinamicamente, di decomposizione (i successori della radice per la versione base e le alternative alla variante principale per la versione applicata a PVSplit). Quella che si intende sviluppare, invece, è una struttura algoritmica assolutamente generale che consenta, a priori, di attuare la distribuzione della ricerca localmente a qualsiasi nodo interno dell'albero di gioco.

La definizione di decomposizione dinamica formulata nel Capitolo 2 non chiarisce alcune proprietà della decomposizione stessa:

• in quale momento della visita è stabilita l'eventuale decomposizione di un nodo (in alternativa alla sua ricerca sequenziale)?

• chi la stabilisce?
• e in base a quali criteri?

5.2.2.2.1 Fasi della ricerca in cui è fissata la decomposizione

Si ritiene esista una sola risposta ragionevole al primo dei tre quesiti: la decomposizione viene decisa esclusivamente nel corso della valutazione del nodo cui essa si riferisce. L'idea è dunque di ritardare l'eventuale decomposizione di un sottoalbero fino al momento in cui si rende necessaria la sua esplorazione.

A chiarimento di questo approccio si consideri l'albero di gioco di Fig. 5.16; il processore P è responsabile della sua valutazione in quanto associato al nodo radice[24]. Si consideri l'ordine di visita depth-first indotto dall'algoritmo $\alpha\beta$; esso stabilisce che i successori della radice siano valutati nell'ordine: N_1, N_2, N_3.

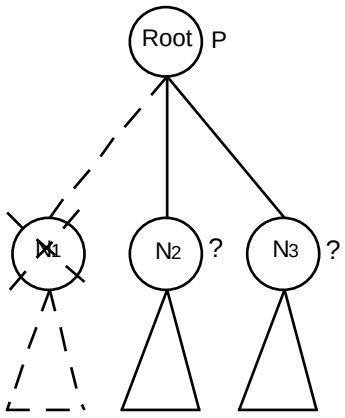


Fig. 5.16 Assegnamento dinamico dei responsabili dei nodi

Si supponga che la visita sia in corso e il sottoalbero relativo a N_1 sia stato già valutato. Nel caso in cui tale valutazione non abbia prodotto un taglio si rende necessaria la visita del sottoalbero N_2: è in questo momento che viene creata l'associazione nodo-processore relativa ad N_2. Un qualche agente (la cui identità è per il momento lasciata imprecisata) stabilisce, su richiesta di P, quale processore sarà responsabile del sottoalbero di N_2; nulla esclude che sia lo stesso P il processore designato. In questa eventualità P verrà impegnato nella valutazione di N_2 e solo quando questa sarà completata verrà presa in considerazione la "sorte" di N_3 (a meno di un eventuale taglio[25]). Se P fosse qualificato come responsabile di tutti i suoi successori, allora il nodo radice sarebbe di fatto non di decomposizione.

Si supponga, invece, che sia fissato per N_2 un responsabile Q diverso da P: la radice diviene un nodo di decomposizione. Non è ragionevole che P si sospenda in attesa che Q completi la valutazione di N_2: che ne sarebbe del parallelismo? È infatti facile dimostrare che in conseguenza di questa soluzione vi sarebbe al più un processore attivo, mentre tutti gli altri sarebbero sospesi in attesa di ricevere un risultato (o privi di attività). L'intuizione corretta è dunque che P si disinteressi del completamento o meno della visita di N_2 e passi a considerare la valutazione del successore seguente (N_3) facendo sì che per esso sia decretato un responsabile ed abbia quindi inizio la sua visita. In generale, dunque, la valutazione del sottoalbero N_3 procede in parallelo a quella di N_2.

I vantaggi di questo approccio emergono evidenti:

• il problema di decomporre la visita di un certo sottoalbero viene preso in considerazione solo se essa è realmente necessaria. Come già spiegato, infatti, la ricerca di alcuni sottoalberi

è soppressa dalle proprietà di taglio dell'algoritmo $\alpha\beta$;

• è favorita l'applicazione di strumenti per il controllo e la regolazione dell'overhead di ricerca e del parallelismo reale dell'algoritmo; si ha così la possibilità di sviluppare algoritmi di decomposizione dove la combinazione delle due proprietà appena elencate sia tale da massimizzarne le prestazioni complessive;

• sono stabilite condizioni per l'applicazione efficace di meccanismi per il bilanciamento del carico: una forte limitazione alla disuniformità della distribuzione del carico è infatti garantita dalla possibilità di impiegare processori inoperosi nel sostenere la ricerca ancora incompleta di alcuni altri.

5.2.2.2.2 Il responsabile della decomposizione

È il momento di chiarire l'identità dell'agente avente il compito di decidere della decomposizione di un nodo, cioè di fissare (su richiesta del suo responsabile) i processori responsabili dei suoi successori. La trattazione non può prescindere da alcuni cenni riguardo il tipo di conoscenza necessaria per prendere questa decisione. Tale conoscenza può essere di duplice natura:

• conoscenza locale e
• conoscenza globale.

Sia il problema di designare il responsabile di un nodo S. Gli attributi di località e globalità della conoscenza sono in riferimento al processore P supervisore di S. Per conoscenza locale si intende pertanto l'insieme di informazioni disponibili localmente ad un supervisore, parte cioè del suo ambiente e quindi accessibili senza alcuna interazione con agenti esterni. Un esempio di conoscenza locale è il livello che il nodo S occupa nell'albero oppure la profondità del sottoalbero di cui esso è radice.

Si supponga per il momento che la designazione del responsabile di un generico nodo S sia effettuata esclusivamente sulla base di conoscenza locale al suo supervisore. In questa ipotesi è immediato indicare in quest'ultimo il misterioso agente destinato alla

nomina del responsabile del nodo S: sono quindi i processori supervisori che autonomamente decidono (tempi e modalità) della decomposizione del sottoalbero loro affidato.

Limitare il tipo di conoscenza alla sola locale offre il vantaggio di escludere interazioni fra moduli e quindi di rendere estremamente efficiente il completamento della scelta del responsabile di un nodo.

Un esempio di informazione globale è, ad esempio, il numero di processori in stato "idle" (cioè che attendono sia loro assegnata la valutazione di un nuovo nodo) oppure il numero di ricerche già distribuite, ma ancora non riconosciute e iniziate dagli esploratori loro assegnati. Perché, in generale, si rende necessario anche questo tipo di conoscenza?

L’assenza di conoscenza globale impedisce al supervisore di avere informazioni sullo stato complessivo della ricerca ed in particolare sull’occupazione o meno degli altri processori. Di conseguenza è impedita l’attuazione di qualsiasi meccanismo per una soluzione dinamica del problema del bilanciamento del carico. Queste considerazioni rendono necessaria anche l’inclusione di questo tipo di conoscenza se non vuole essere pregiudicata l’efficienza complessiva della ricerca parallela.

La soluzione generale che include il ricorso ad entrambe le forme di conoscenza prevede che la designazione del responsabile di un nodo S avvenga in due fasi:

• sia P il supervisore di S; la conoscenza locale a P è incorporata in un test booleano il quale stabilisce se l’esploratore di S deve essere lo stesso P. Questo primo stadio assolve dunque la funzione di filtrare secondo un certo criterio (inserito nel test) quei nodi la cui valutazione non si vuole distribuire indipendentemente dallo stato complessivo della ricerca;

• nel caso in cui il test locale a P dia esito negativo si deve ricorrere all’analisi dello stato globale della ricerca e in base a questa finalmente fissare l’esploratore di S (nulla esclude che esso possa essere P).

La seconda fase rende necessaria l’interazione fra i processori nella decisione dinamica della

decomposizione o meno di un nodo. Come avviene la gestione della conoscenza globale e quali interazioni inter-modulo sono necessarie alla sua distribuzione?

• in un sistema di programmazione parallela ad ambiente locale la scelta sarebbe forzata: un modulo con sole finalità di gestore mantiene nel suo ambiente tutta la conoscenza "logicamente" globale. Esso interagisce con i moduli di ricerca per aggiornare (in modo consistente) tali informazioni. La sua funzione principale è tuttavia applicare il criterio "globale" di attribuzione di un processore all'esplorazione di un nodo quando ciò è richiesto dal supervisore di quest'ultimo: si può presumere che quest'attività sia completata dalla comunicazione di due messaggi, diretti rispettivamente all'esploratore designato (con la descrizione della ricerca lui assegnata) e al supervisore stesso (contenente l'identità dell'esploratore).

• in un sistema Linda, invece, alla gestione centralizzata delle informazioni globali si preferisce una cooperazione alla pari mediata dall'organizzazione delle informazioni in una struttura dati distribuita (l'efficacia di questo modello è stata dimostrata nell'implementazione dell'algoritmo base). La soluzione che verrà presentata è basata su questo modello.

È dunque previsto che tutta la conoscenza globale sia memorizzata in strutture dati distribuite e "gestita" dagli stessi processori destinati alla ricerca. L'idea che si vuole sviluppare è che il test locale del supervisore P di un nodo N sia generalizzato e sia basato anche su una porzione della conoscenza globale: esso stabilisce definitivamente se l'esplorazione di N deve essere distribuita (cioè assegnata ad un processore diverso da P) o deve proseguire sequenzialmente (P diviene anche suo esploratore). Nella prima delle due eventualità il lavoro di ricerca di N viene depositato in una struttura dati distribuita. Si osservi che la designazione dell'esploratore di N non è ancora avvenuta: essa sarà attuata implicitamente quando uno dei processori, divenuto inoperoso, rimuoverà per primo la

richiesta di ricerca di N dalla struttura distribuita appena detta. Sia Q tale processore; assunto il ruolo di esploratore di N esso deve quindi completarne la valutazione ed infine comunicare il risultato a P. Il tipo di relazione che è stata stabilita dinamicamente fra P e Q è la stessa che lega il modulo master e uno dei moduli worker nel paradigma di programmazione parallela omonimo. Si osservi che il nodo di cui Q è divenuto esploratore può diventare anch'esso di decomposizione; Q può quindi rivestire, contemporaneamente al ruolo di worker, anche quello di master della distribuzione della ricerca al livello inferiore.

Localmente ad un nodo N di decomposizione viene dunque creato quell'insieme di strutture dati distribuite ed interazioni inter-modulo che caratterizzano il modello master-worker: il ruolo del master è rivestito dal supervisore di N, mentre quello di worker dagli esploratori (≠ dal supervisore) dei successori di N. Dei moduli coinvolti in questa interazione solamente il master rimane invariato; al contrario, la struttura di worker si evolve dinamicamente in quanto ogni worker è legato ad un nodo di decomposizione solo per il tempo della valutazione di uno dei suoi successori. Esso può cioè assumere il ruolo di worker anche in decomposizioni diverse che procedono in parallelo[26], passando dal servizio di un master all'altro; completata la valutazione di un nodo, infatti, la scelta del successivo da esplorare fra quelli presenti nelle strutture di lavori relative a tutte le decomposizioni in corso è nondeterministica e non può essere influenzata dalla storia dei precedenti servizi di quel processore. Si può pensare che dal punto di vista del generico processore (worker) i lavori presenti in strutture distinte fanno parte di un'unica struttura generale dove sono indistinguibili.

Un nodo N diviene di decomposizione nel momento in cui avviene la prima distribuzione della ricerca di uno dei suoi successori; è in questo momento che la ricerca di N passa da sequenziale a parallela e sono create logicamente le strutture di lavori e di risultati necessarie alla sua implementazione. La

rimozione logica di quest’ultime avviene quando è completata la valutazione di N.

5.2.2.2.3 I criteri di decomposizione

Ancora uno dei tre interrogativi iniziali deve essere risolto: in base a quali criteri è decisa la decomposizione o meno di un nodo dell’albero di gioco?

Considerazioni precedenti hanno dimostrato che la risposta a questa domanda è legata alla soluzione di un altro sottoproblema: stabilire se un nodo di cui un processore è supervisore deve essere esplorato dallo stesso processore oppure distribuito ad uno diverso. Questa decisione, operata autonomamente dal supervisore, può essere descritta da un’unica funzione booleana; quest’ultima incorpora il criterio di decomposizione, cioè l’insieme di regole che determina i nodi di decomposizione e i particolari della loro decomposizione (cioè quali dei successori sono distribuiti nella rispettiva esplorazione).

Il criterio di decomposizione rappresenta la parte caratterizzante di un algoritmo di decomposizione: l’avere confinato questa componente in un’unica funzione booleana significa avere definito una struttura software estremamente flessibile, capace di ospitare facilmente qualsiasi algoritmo della classe in analisi (sia esso di tipo statico o dinamico).

Come noto, l’applicazione del criterio di decomposizione da parte di un supervisore richiede conoscenza che può essere locale o globale a tale processore. L’acquisizione della conoscenza globale è, in termini di efficienza, molto più costosa. Poiché il numero di invocazioni della funzione booleana di test è elevato (pari, in una ricerca, al numero di nodi visitati), un criterio di decomposizione ragionevole deve limitare quanto possibile la necessità delle informazioni globali. L’idea generale è decomporre il criterio di decomposizione in due sottocriteri, rispettivamente applicati alle porzioni locale e globale della conoscenza di cui esso necessita. Il criterio "locale" ha la funzione di filtrare quei nodi che, indipendentemente dallo stato globale della ricerca, non è conveniente esplorare in parallelo; l’acquisizione della

conoscenza globale e l’applicazione del sottocriterio relativo hanno luogo solo nell’ipotesi che il sottocriterio locale "approvi" l’eventuale distribuzione del sottoalbero in esame.

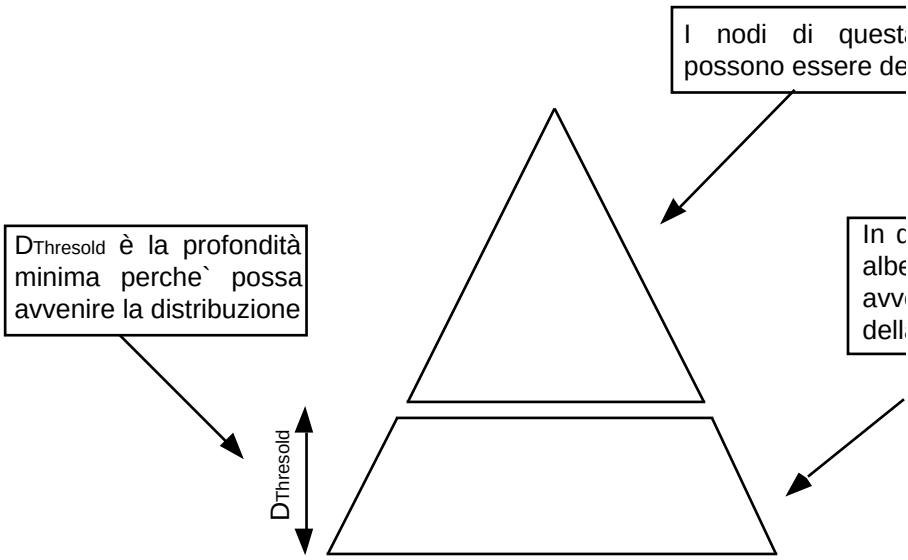


Fig. 5.17 Sottocriterio di decomposizione basato sulla dimensione del sottoalbero

Sono ora elencati alcuni esempi di sottocriteri locali:

• sottocriteri basati sulle dimensioni del sottoalbero: vengono esclusi dalla distribuzione sottoalberi di piccole dimensioni in quanto l’entità del degrado indotto dalla ricerca parallela (in particolare quello dovuto alle comunicazioni) fa preferire in questi casi la visita sequenziale. La determinazione della dimensione di soglia non è realisticamente ottenibile con metodi analitici perché ciò richiederebbe un troppo complesso studio quantitativo delle forme di degrado; la tecnica più semplice è quella empirica del "trial-and-error" che procede per approssimazioni successive della soluzione migliore.

La misura più corretta delle dimensioni di un albero è il numero di nodi; purtroppo questa informazione non può essere nota prima che la visita $\alpha\beta$ abbia inizio. È comunque possibile stimare con buona approssimazione il numero di nodi che saranno visitati conoscendo la

profondità dell’albero e la larghezza della finestra $\alpha\beta$ iniziale [Bau78b]: il costo del calcolo di tale stima non è comunque trascurabile poiché include operazioni come l’elevamento a potenza e la radice.

Una soluzione più semplice è approssimare le dimensioni dell’albero con la sua profondità [Eli90]: ciò induce una partizione dei nodi in due sottoinsiemi come illustrato in Fig. 5.17.

• il criterio "young-brothers-wait" [FMMV89]: questo è il primo di una serie di criteri finalizzati a ridurre l’overhead di ricerca; tali criteri hanno significato solo nell’ipotesi di alberi (almeno parzialmente) ordinati. Il criterio stabilisce che il primo successore di un nodo deve essere completamente valutato prima che possa avere inizio la visita dei successivi. Il dato locale necessario per applicare questo metodo è l’ordine di visita I (rispetto ai suoi "fratelli") del nodo N in esame: se I=1, allora il nodo deve essere valutato dal suo supervisore (e quindi non distribuito). Si osservi come in questa eventualità è garantito che la decomposizione dei fratelli sia presa in considerazione solo dopo l’avvenuta valutazione di N.

Una naturale generalizzazione di questo metodo è che sia impedita la distribuzione dei primi K ($K \geq 1$) successori; la valutazione di questi nodi avverrà dunque in sequenza. In questo caso la condizione sufficiente ad impedire la distribuzione del nodo è: $I>K$, dove I è l’indice ordinale del nodo cui è applicato il criterio.

La giustificazione del criterio è che se i nodi interni dell’albero di gioco sono ordinati, allora con elevata probabilità la migliore mossa è individuata da uno dei nodi più a sinistra; visitare completamente i sottoalberi relativi a tali nodi prima di affidare i successivi ad altri processori significa ottenere una finestra $\alpha\beta$ migliore per la ricerca di quest’ultimi.

• limitazione del numero di successori visitati in parallelo: si intende limitare il massimo parallelismo ottenibile localmente alla decomposizione di un nodo. Lo scopo di questo approccio è anch’esso di limitare il problema dovuto al fatto che le ricerche di successori successivi non usufruiscano dei

risultati di visite iniziate precedentemente, ma ancora incomplete. Anche per questo controllo è necessario un dato esclusivamente locale al supervisore: il numero di nodi la cui ricerca è stata da esso distribuita, ma dei quali non è stato ancora restituito il risultato.

I seguenti criteri, invece, sono basati su conoscenza globale:

• limitazione del numero di lavori in attesa di essere eseguiti. Si consideri la struttura logica complessiva di tutti i lavori (come appare ad un generico worker): il criterio intende limitare il numero massimo di lavori che possono essere ospitati in essa; lo scopo è ridurre l'overhead di sincronizzazione. Infatti se le richieste di servizio fossero in numero eccessivo rispetto al numero di serventi, ciò determinerebbe un'attesa eccessiva da parte dei richiedenti. Nel caso del presente algoritmo tale attesa sarebbe concentrata nella fase finale della decomposizione di un nodo quando, stabilita per tutti i successori la distribuzione o la visita sequenziale (già completata), il supervisore si sincronizza nella ricezione di tutti i risultati non ancora comunicati. Un valore tipico per la dimensione massima della struttura di lavori è il numero totale di processori. L'informazione globale coinvolta in questo criterio è il numero di nodi presenti nell'agenda generale. L'implementazione in Linda prevede a riguardo la definizione di una variabile distribuita con funzione di contatore, incrementata dal supervisore che distribuisce un nuovo lavoro e decrementata dal worker che completa l'estrazione di uno di essi.

• distribuzione solo se esiste almeno un nodo inoperoso (idle): il criterio previene l'eccessiva permanenza di un lavoro in agenda e quindi riduce anch'esso l'overhead di sincronizzazione poiché minimizza il tempo che intercorre fra la domanda di un servizio e il completamento di questo. In questo caso l'informazione globale è il numero di processori inoperosi; la relativa implementazione in Linda è analoga a quella descritta per il contatore di lavori in agenda.

5.2.2.2.4 Un chiarimento delle finalità dell’algoritmo

Sono state finalmente chiarite tutte le caratteristiche generali di un algoritmo di decomposizione dinamica parametrico rispetto al criterio di decomposizione; sono state inoltre fornite alcune indicazioni sulla sua implementazione in Linda.

Si osservi che l’implementazione su questa struttura algoritmica degli algoritmi base e PVSplit sarebbe immediata data la semplicità del criterio di decomposizione di entrambi; la performance degli algoritmi così ottenuti sarebbe tuttavia inferiore a quella delle implementazioni presentate nei paragrafi precedenti: perché?

La flessibilità dell’algoritmo generale di decomposizione non può essere sinonimo di efficienza: il requisito di generalità con cui esso è stato concepito ha impedito di tenere conto di ottimizzazioni possibili solo per un certo tipo di criteri di decomposizione. La struttura software progettata intende essere essenzialmente uno strumento di ricerca che permetta di stimare con immediatezza le prestazioni di nuovi criteri di decomposizione, senza la necessità di dover ogni volta ridefinire la struttura di interazioni inter-processo. È così suggerito un approccio allo sviluppo di un algoritmo parallelo di ricerca $\alpha\beta$ che prevede due fasi consecutive:

• una prima fase di selezione del criterio di decomposizione più promettente e, successivamente

• una fase di specializzazione dell’algoritmo generale che tenga conto delle caratteristiche del criterio di decomposizione emerso dalla prima fase in modo da massimizzarne l’efficienza.

Quale esemplificazione dei concetti appena espressi saranno messe in evidenza alcune delle ottimizzazioni già viste negli algoritmi base e PVSplit che non possono essere applicate nell’algoritmo generale di decomposizione dinamica.

5.2.2.2.5 La descrizione del lavoro di ricerca.

Gli algoritmi di decomposizione statica presentati nei paragrafi precedenti

prevedevano un insieme di nodi di decomposizione limitato e noto a priori; queste caratteristiche avevano permesso di identificare un nodo, la cui valutazione doveva essere distribuita, attraverso il solo indice posizionale. Questa ottimizzazione era applicata anche nei casi in cui l'ordinamento dei nodi interessati alla distribuzione fosse variabile dinamicamente e quindi obbligato ad essere memorizzato in strutture globali (cfr. 5.2.1.3): essa restava infatti conveniente in virtù del numero esiguo di tali nodi.

L'algoritmo generale di decomposizione dinamica, invece, prevede che uno qualsiasi dei nodi dell'albero di gioco possa divenire di decomposizione. Questa proprietà fa sì che un nodo la cui esplorazione debba essere distribuita possa essere identificato esclusivamente dalla sua rappresentazione completa; l'identificazione di un nodo attraverso un sistema di indici posizionali richiederebbe in generale la definizione di una struttura distribuita di dimensioni proporzionali a quelle dell'albero di gioco. In conseguenza le operazioni di deposito e prelievo di un lavoro acquisiscono un costo relativo molto elevato e sono quindi origine di un maggiore degrado di comunicazione. L'incidenza di tale overhead sarà più sensibile in quelle architetture parallele dove la comunicazione fisica fra processori è meno efficiente (ad es. reti di workstation).

5.2.2.2.5.1 Segnature, alberi di segnature e gestione dei tagli

L'algoritmo generale prevede che più nodi di decomposizione possano essere esplorati in parallelo; questa proprietà rende necessario, per un generico worker, identificare la struttura di risultati in cui depositare l'esito del suo lavoro. Si osservi che quale identificatore di tale

struttura non è sufficiente indicare il nome del processore master che ha generato quel lavoro: l'ambiguità deriva dal fatto che un processore può assumere il ruolo di master in decomposizioni distinte che procedono in parallelo (Fig. 5.18).

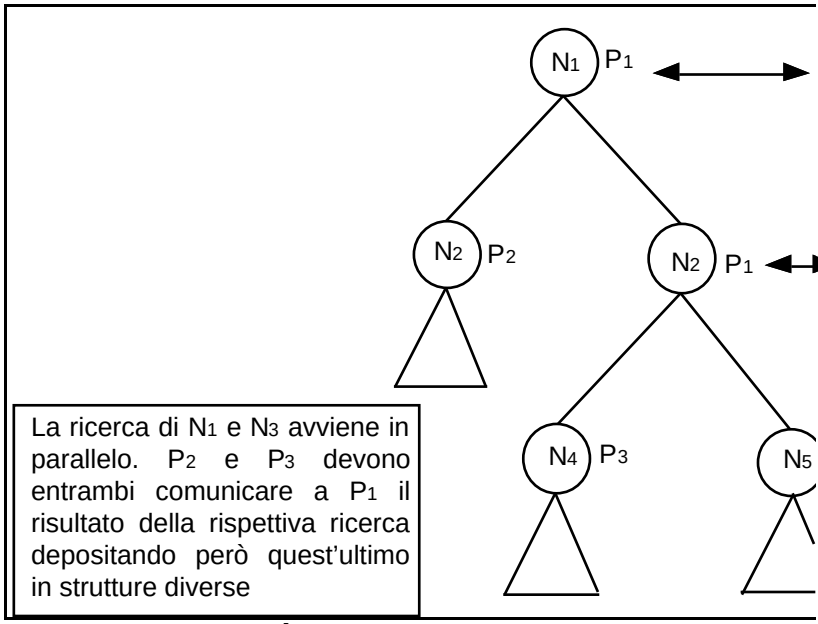


Fig. 5.18 Ambiguità nella determinazione della struttura di risultati

La soluzione corretta è dunque associare un identificatore unico (segnatura) a ciascun nodo di decomposizione N. Tutti i successori di N ereditano tale segnatura, anche quelli la cui ricerca non è distribuita. Nel caso in cui la visita di uno dei successori di N produca un taglio, allora tutte le ricerche in parallelo con la stessa segnatura di N sono terminate prematuramente; in tale modo si previene che siano portate a termine esplorazioni che il taglio ha ormai reso inutili.

```
void init_signature ()
{
out ("sig",0);
}
int new_signature ()
{
int sig;
```

```
in ("sig",?sig);
out ("sig",sig+1);
return (sig);
}
```

Fig. 5.19 Segnatura

In Fig. 5.19 sono presentate le funzioni in Linda che implementano la cooperazione fra i moduli per l'attribuzione della segnatura ad un nuovo nodo di decomposizione.

Una proprietà dell'algoritmo generale è il subappalto della ricerca; esso può avvenire, in generale, in qualunque livello dell'albero. Questa caratteristica consente anche l'annidamento del subappalto, limitatamente alla profondità dell'albero ed al numero di processori.

Ogni nuovo nodo di decomposizione è accompagnato da una nuova segnatura; può dunque accadere, in virtù del subappalto della ricerca, che nodi di uno stesso sottoalbero presentino segnature differenti. Quando viene generato un taglio al top-level di detto sottoalbero, tutte le ricerche in parallelo (contenute in esso) dovrebbero essere arrestate, anche se aventi segnature differenti.

L'implementazione di tale idea è basata sulla gestione dell'albero delle segnature: è una struttura dati distribuita contenente la relazione gerarchica fra le segnature associate alle decomposizioni in corso. La radice di questo albero è la segnatura della radice dell'albero di gioco.

Si supponga, ad esempio, che la ricerca abbia raggiunto un nodo che ha ereditato, al momento della sua generazione, la segnatura S del padre; si supponga che tale nodo divenga di decomposizione: ad esso viene associata una nuova ed unica segnatura T. In questo caso all'albero delle segnature è aggiunto un nodo con etichetta T come successore del nodo con etichetta S. Quando viene prodotto un taglio da una ricerca con segnatura S, allora sono terminate in cascata tutte

le ricerche parallele con segnatura S o con qualsiasi discendente della segnatura S nell’albero delle segnature.

In Fig. 5.20a è mostrato un ipotetico albero di gioco. Le linee tratteggiate individuano sottoalberi distribuiti ad altri processori; l’etichetta dei nodi rappresenta invece la segnatura locale. L’albero di gioco contiene quattro decomposizioni locali. Durante la decomposizione (a) sono distribuiti i lavori relativi ai sottoalberi più a sinistra; le altre decomposizioni prevedono invece la distribuzione di un solo lavoro. Fig. 5.20b presenta l’albero delle segnature corrispondente a tale scenario.

Si supponga che sia generato un taglio al livello della decomposizione (a) a causa della valutazione del sottoalbero più a sinistra: tutte le ricerche con segnature 1, 2, 3 e 4 saranno terminate. Si ipotizzi, invece, che il taglio sia prodotto al livello della decomposizione (b) dalla visita del sottoalbero più a destra: in questo caso saranno "uccise" solamente le segnature 2 e 4.

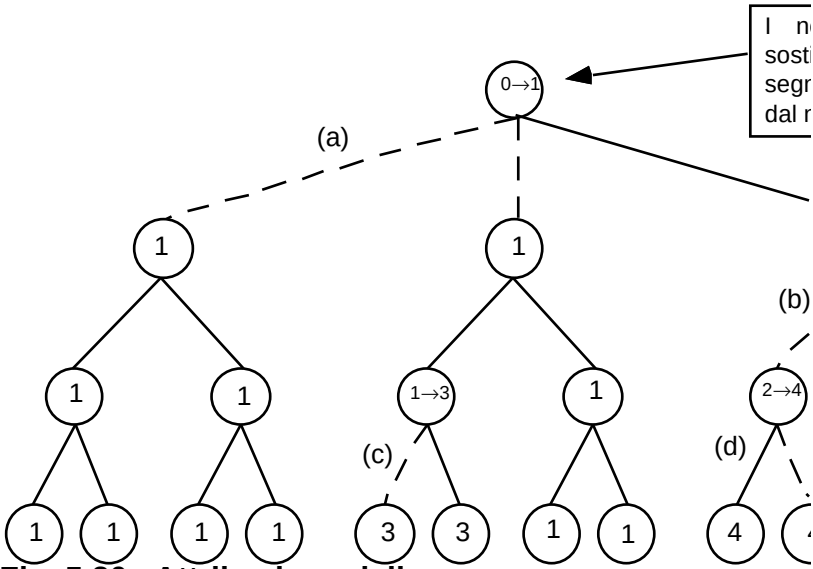


Fig. 5.20a Attribuzione della segnatura

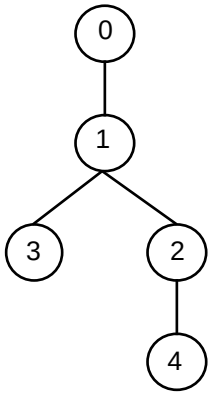


Fig. 5.20b Albero delle segnature

In Fig. 5.21 è descritta l'implementazione in Linda dell'albero delle segnature, delle sue operazioni di aggiornamento e, ad esse legata, della gestione della terminazione prematura delle ricerche provocata dai tagli $\alpha\beta$.

Si osservi che l'implementazione come struttura dati distribuita dell'albero delle segnature è più semplice di quella suggerita nel paragrafo 3.4.1.3.2 per strutture ad albero. In questo caso l'albero non è memorizzato per nodi, ma per archi; esso è infatti definito da tuple con struttura:

```
("sig_tree", father, son)
```

```
int killed (sig)
int sig;
{
return (!rdp("no_cut",sig));
}
int add_signature (int master_sig, int
new_sig)
{
out ("sig_tree", master_sig,
worker_sig); /* aggiungo un nuovo nodo
*/
if (killed (master_sig)) /* la ricerca
è stata uccisa? */
       {
/* il processo deve rimuovere esso
stesso il nuovo nodo dell'albero delle
segnature, a meno che ciò non sia già
avvenuto ad opera del processo che ha
prodotto il taglio e cui spetta la
gestione della terminazione dei
processi ad essa interessati */
       inp ("sig_tree", master_sig,
worker_sig);
       return (false);
       }
else
```

```
       return (true); /* la creazione
del nuovo nodo "di segnatura" ha avuto
successo */
}
/* la funzione kill_subtree ha per
argomento la segnatura della
decomposizione al livello della quale è
stato prodotto il taglio; la sua
funzione è di rimuovere il nodo
dell'albero delle segnature relativo a
tale segnatura e di invocare la
funzione ricorsiva kill_sons che
rimuove dallo stesso albero le
segnature più in profondità */
int kill_subtree (int mysig)
{
void kill_sons ();
/* è possibile che l'occorrere di tagli
simultanei (quando relativi a
decomposizioni annidate) faccia sì che
più processi concorrano alla rimozione
di un nodo dell'albero delle segnature;
il controllo che segue garantisce la
correttezza di tale rimozione */
if (inp ("sig_tree", ?int, mysig))
       {
       in ("no_cut", mysig); /* la
presenza di questa tupla comunica che
tutte
       le ricerche con segnatura mysig
devono essere
       terminate prematuramente */
       kill_sons (mysig);  /* applica la
terminazione a tutte le segnature che
       discendono da mysig */
       return (true);
       }
else
       return (false);
}
void kill_sons (int sig)
{
int sig_son;
while (inp ("sig_tree", mysig,
?sig_son))
       {
       in ("no_cut", sig_son);
       kill_sons(sig_son);
       }
}
```

Fig. 5.21 Gestione dell'albero delle segnature

A completamento della descrizione della gestione dei tagli deve essere precisato che:

• la richiesta di terminazione forzata di una ricerca viene rilevata dal processo che la sta eseguendo testando (funzione killed) la presenza o meno di una particolare tupla (con etichetta "no_cut") il cui contenuto è la segnatura della propria ricerca; la frequenza di tale

test dipende dal compromesso ottimale fra il costo di un test e i vantaggi (in termini di overhead di ricerca) apportati da un pronto rilevamento della richiesta di terminazione;

• la rimozione del sottoalbero delle segnature interessato da un taglio e di tutte le relative tuple con etichetta "no-cut" è a cura del processo che lo ha prodotto. Può accadere che siano generati simultaneamente due o più tagli al livello di decomposizioni che sono annidate: quale dei processi deve rimuovere il sottoalbero di segnature più interno?

Sarà il primo di essi che rimuoverà il nodo ("sigtree",father,sig), dove sig costituisce la segnatura di tale sottoalbero: è l'indivisibilità dell'operazione Linda di rimozione di tuple (in) a garantire la correttezza di questa soluzione.

5.2.2.2.6 I risultati sperimentali

L'algoritmo è stato sperimentato applicando ad esso diversi criteri di decomposizione dinamica; in questa sede sono presentati i dati statistici ricavati per due di essi:

• criterio A: il criterio locale stabilisce che non possa essere distribuita la visita di sottoalberi con profondità≥2 ed applica la regola "young-brothers-wait"; il criterio globale, invece, prevede che possa essere generato un lavoro solo se esiste un processo in stato "idle" e non siano già presenti in agenda (P-1) lavori, dove P è il numero di processi.

• criterio B: rispetto al criterio A è disabilitata la gestione dell'albero delle segnature ed è modificato il criterio globale prevedendo che un lavoro possa essere deposto in agenda anche se non esiste nessun processo inoperoso.

In Tab. 5.9 e Tab. 5.10 sono riportati i risultati sperimentali relativi a tali criteri.

nr. proc	T (sec)	N (nodi)	S	E	Fpm	SO	VR (nodi/sec)	Svr
1	8301	5347545	---	---	---	---	644	---
3	3797	6206534	2.19	0.73	0.85	0.16	1635	2.54
5	2634	6829036	3.15	0.63	0.82	0.28	2593	4.02

7	2133	7407997	3.89	0.56	0.77	0.39	3473	5.39
9	1878	7810188	4.42	0.49	0.74	0.46	4159	6.46
11	1699	8205839	4.89	0.44	0.70	0.53	4830	7.50

Tab. 5.9 Criterio A: sperimentazione

nr. proc	T (sec)	N (nodi)	S	E	Fpm	SO	VR (nodi/sec)	Svr
1	8301	5347545	---	---	---	---	644	---
3	4629	6476838	1.79	0.60	0.84	0.21	1399	2.17
5	2798	7280335	2.97	0.59	0.88	0.36	2602	4.04
7	2223	8133582	3.73	0.53	0.89	0.52	3659	5.68
9	1972	8541944	4.21	0.47	0.89	0.60	4332	6.72
11	1663	8689459	4.99	0.45	0.89	0.62	5225	8.11

Tab. 5.10 Criterio B: sperimentazione

Sebbene differenti in pochi aspetti, i due criteri evidenziano prestazioni fortemente contrastanti. Il criterio A è caratterizzato da un overhead di ricerca più contenuto giustificato dalla gestione dell'albero delle segnature; il vantaggio della riduzione di questa forma di degrado permette al criterio A di essere più efficiente del criterio B in presenza di un numero esiguo di processori. All'aumentare di quest'ultimi, però, il criterio A evidenzia un crollo del fattore di produzione: qual è la motivazione? La condizione "distribuire solo in presenza di almeno un processore inoperoso", nel tentativo di ridurre il tempo di risposta alla richiesta di un generico lavoro, riduce il numero di lavori distribuiti e origina con elevata frequenza situazioni in cui l'agenda è vuota e vi sono processori in attesa di lavoro. Questo dato statistico rivela un tipico errore di progetto delle interazioni fra moduli: la frequenza di completamento dei servizi è troppo elevata rispetto a quella della loro richiesta.

Il criterio B rilascia il vincolo appena discusso: l'effetto è di aumentare il numero medio di lavori in agenda e quindi stabilire un minore accoppiamento fra master e worker. I benefici di questa variazione si avvertono all'aumentare del numero di processori: si osservi come il fattore di produzione si attesti su valori molto elevati e pressoché costanti (89% con 11 processori).

L'algoritmo parallelo che emerge dal criterio B è rappresentativo della classe di algoritmi ad elevata interazione inter-modulo; la notevole

decomposizione dell'albero di gioco che esso induce determina infatti una fitta rete di comunicazioni fra i processi relative alla distribuzione dei lavori ed al recupero dei rispettivi risultati. Per questa classe di algoritmi è interessante analizzare l'incidenza delle comunicazioni sulla performance complessiva.

Si considerino i valori del fattore di produzione medio F_{pm} in Tab. 5.10: essi non indicano la reale produttività di un generico processo perché comprensivi anche della porzione di tempo che li ha visti impegnati nelle comunicazioni. Il dato statistico F_{pm} deve dunque essere scorporato nelle due componenti F_{pmr} (produzione reale) e OC (overhead di comunicazione) adottando il metodo illustrato nel paragrafo 5.2.1.4.1; in Tab. 5.11 è presentato il dettaglio di tali statistiche.

nr. proc.	Fpm	Fpmr	OC
3	0.85	0.84	0.01
5	0.88	0.80	0.08
7	0.89	0.80	0.09
9	0.89	0.75	0.14
11	0.89	0.73	0.16

Tab. 5.11 Criterio B: componenti del fattore medio di produzione

I dati di Tab. 5.11 parlano di un overhead di comunicazione tutt'altro che trascurabile; su esso incidono l'elevato numero di interazioni inter-processore che caratterizzano l'algoritmo e l'inefficienza con cui esse sono tipicamente eseguite in un'architettura parallela di tipo rete locale. La natura dell'algoritmo impedisce l'utilizzo delle primitive Linda nel limitare il numero di informazioni che devono essere fisicamente scambiate fra i processori. In proposito sono state viste tecniche (come l'uso di tuple private) che favoriscono l'accesso a informazioni globali già presenti nella memoria locale del processore e che quindi non richiedono interazioni fisiche inter-processore. Tuttavia esse non sono applicabili nell'algoritmo in analisi: l'informazione il cui costo di comunicazione è maggiore, la

descrizione di un lavoro, non è mai ricevuta dallo stesso processo che la ha generata e deve essere sempre trasferita fisicamente tra due processori.

5.3 Conclusioni

In questo capitolo abbiamo studiato come utilizzare Linda nella realizzazione di alcuni algoritmi di decomposizione dell'albero di gioco.

A partire da un algoritmo relativamente semplice, l'algoritmo base, sono state applicate ad esso nuove idee che ne hanno progressivamente migliorato le prestazioni.

Nel corso della discussione è stato più volte verificato che in generale non esiste un unico modo di attuare in Linda tali miglioramenti, ma ne esiste comunque uno più efficiente.

Ad esempio è stato indicato dagli esperimenti che per la classe di problemi studiata il modello di cooperazione alla pari, quando attuabile, è preferibile al modello master-worker, anche se, tuttavia, la differenza in termini di prestazioni fra le due scelte si attenua all'aumentare del numero di processori.

Abbiamo inoltre "scoperto" che in certe situazioni è la stessa implementazione dello spazio delle tuple a condizionare le scelte del programmatore di applicazioni: in 5.2.1.4.1 è stata proposta una tecnica di programmazione, basata sull'uso di tuple private, volta a ridurre il costo di condivisione di una struttura dati in un sistema Linda con "in distribuita" quando tale struttura è acceduta più frequentemente in lettura piuttosto che in aggiornamento.

Abbiamo utilizzato quale misura delle prestazioni degli algoritmi un parametro standard: lo speedup S. Abbiamo verificato che raggiungere il limite ideale per questo parametro (N se N è il numero di processori impegnati della ricerca) è compito assai difficile. Nel corso degli esperimenti è stata rilevata la presenza di 3 principali motivi che hanno contribuito al degrado delle prestazioni:

- il costo delle comunicazioni fra processori (overhead di comunicazione)
- la parziale condivisione dei risultati intermedi (overhead di ricerca)
- la distribuzione non uniforme del carico di lavoro (overhead di sincronizzazione)

È stato appurato il forte legame esistente fra queste forme di degrado: la riduzione di una ha sempre determinato l'aumento di un'altra; l'algoritmo base con decomposizione dinamica al top-level (cfr. Tab. 5.7), ad esempio, è quello che ha garantito il migliore bilanciamento del carico (Fpm=90% con 11 processori), ma ha anche generato l'overhead di ricerca più disastroso (SO=108% con 11 processori). La conoscenza di questa realtà ci ha guidato nello sviluppo di soluzioni algoritmiche mirate ad ottenere il migliore bilanciamento (trade-off) fra le forme di degrado, cioè a minimizzare la loro incidenza complessiva e non di una sola di esse.

I risultati sperimentali hanno dimostrato che tale situazione ottimale è favorita dall'algoritmo PVSplit quando è combinato con il concetto di condivisione dello score (S=5.57 con 11 processori; cfr. Tab. 5.6b) o con quello di distribuzione dinamica (S=5.58 con 11 processori; cfr. Tab. 5.8).

I valori per lo speedup appena indicati assumono significato solo se confrontati con quelli ottenuti da altri ricercatori nella sperimentazione di algoritmi simili.

In [MaOlSc86] viene segnalato uno speedup per PVSplit di 3.66 con una rete di 5 workstation misurato in base alla esplorazione fino a profondità 7-ply delle 24 posizioni di Bratko-Kopec; lo speedup da noi ottenuto per PVSplit con lo stesso numero di processori è 3.49 (= 3.75 se si considera il miglioramento indotto dalla condivisione dello score; cfr. Tab. 5.6a e 5.6b). Tale valore è molto soddisfacente e testimonia che l'implementazione in Linda della classe di problemi da noi studiati può garantire prestazioni apprezzabili. Ad avvalorare tale affermazione si deve tenere in considerazione il fatto che i nostri esperimenti sono stati eseguiti con profondità di ricerca 5-ply. È stato infatti verificato che lo speedup migliora all'aumentare della profondità di ricerca [MaOlSc86]; il motivo di tale relazione è che i vari tipi di degrado delle prestazioni introdotti dal parallelismo sono maggiormente dissipati all'interno di ricerche più durature e quindi hanno minore impatto sulle prestazioni.

In [Sch89] è riportato uno speedup per DPVSplit di 4.78 ottenuto con 7 processori (ricerca a profondità 7-ply). Lo speedup da noi ottenuto per lo stesso algoritmo, con identico numero di processori e ricerca a 5-ply, è 4.54 (cfr. Tab. 5.8).

In conclusione l'implementazione in Linda e su rete locale dei "migliori" algoritmi di decomposizione statica determina prestazioni confrontabili con quelle ottenute per gli stessi algoritmi in altri ambienti di ricerca.

La realizzazione in Linda di algoritmi con distribuzione dinamica della ricerca ha rivelato prestazioni deludenti (cfr. Tab. 5.9 e 5.10). Tale inefficienza è intrinseca in questa classe di algoritmi o è stata originata dall'ambiente in cui essi sono stati valutati?

Gli eccellenti risultati ottenuti dalla sperimentazione di tali algoritmi sulla base di altre architetture parallele appaiono confermare questa seconda ipotesi:

• in [Eli90] è presentato Oracol, un solutore di problemi di scacchi il cui algoritmo di ricerca è parallelo con decomposizione dinamica dell'albero di gioco. Il massimo speedup riportato è 5.5 ottenuto su un'architettura multiprocessore con memoria condivisa e costituita da 10 processori.

• in [FeMyMo92] è presentato un algoritmo di decomposizione dinamica basato sul criterio "young-brothers-wait". La sua realizzazione, avvenuta su un sistema di transputer composto di

256 processori, ha garantito uno speedup sorprendente: 126 (ricerca a 8-ply).

Questi risultati sperimentali suggeriscono che l'implementazione degli algoritmi di decomposizione dinamica può generare prestazioni notevoli, ma solo in architetture con memoria condivisa o distribuite con grosso accoppiamento fra i processori. In questi ambienti, infatti, il costo delle interazioni interprocessore è molto minore rispetto a quello di una rete locale e sono quindi più adatte per ospitare applicazioni a grana piuttosto fine come gli algoritmi di decomposizione dinamica.

Sarebbe interessante verificare su queste architetture parallele (ed in particolare su quelle a parallelismo massiccio):

• l'atteso miglioramento delle prestazioni dell'algoritmo generale di decomposizione dinamica da noi proposto

• la convergenza asintotica (all'aumentare dei processori) prevista in [MaOlSc86] verso un valore costante dello speedup per i metodi di decomposizione statica.

Capitolo 6
La distribuzione della conoscenza

6.1 Introduzione

I giocatori artificiali di scacchi che usano più processori hanno normalmente la seguente caratteristica: nella scelta della mossa tutti i processori cooperano per completare nel modo più efficiente la visita dell'albero di gioco associato alla posizione corrente. In sostanza avviene una distribuzione della ricerca fra istanze indistinguibili di uno stesso giocatore artificiale (Capitolo 5).

Il tipo di giocatore parallelo che descriviamo in questo capitolo, invece, si fonda su un approccio differente: ogni processore costituisce un'istanza di ricerca dell'albero di gioco indipendente dalle altre. Vengono quindi condotte in parallelo tante ricerche quanti sono i processori. La cooperazione fra le istanze avviene a posteriori, quando cioè ognuna di esse ha proposto la sua mossa migliore e viene allora applicata una regola (denominata criterio di selezione) per la scelta finale di una fra tutte le mosse proposte.

Ciò che di fatto è stabilita da questo nuovo approccio è una distribuzione della conoscenza. Le istanze di ricerca sono infatti tutte diverse in quanto incorporano una differente conoscenza del dominio di applicazione.

Il concetto di conoscenza deve essere chiarito e messo in relazione alla classe di problemi in analisi; Berliner ha proposto una classificazione in due categorie della conoscenza applicata a problemi di ricerca su alberi di gioco [Ber82]:

• conoscenza dirigente: è usata per guidare la ricerca e suggerire l'ordine in cui i discendenti di un nodo devono essere esaminati. In base a questa definizione si può intendere che istanze con differenti algoritmi di visita dell'albero di gioco presentano una differente conoscenza dirigente.

• conoscenza terminale: è usata al livello dei nodi terminali per indicare quanto è desiderabile raggiungere una certa posizione. Due istanze differiscono dunque nella rispettiva conoscenza terminale se hanno funzioni di valutazione diverse.

Il nuovo approccio che sarà descritto si distingue per la sua pressoché assoluta originalità: solamente in [Alt91], infatti, è presentato lo studio di un giocatore parallelo basato su una distribuzione "rudimentale" della conoscenza. In esso ciascuna istanza di ricerca è costituita da un giocatore artificiale commerciale; il criterio di selezione è di tipo a maggioranza (o democratico): la mossa che ha ottenuto il maggior numero di proposte è quella che sarà effettivamente giocata.

Tale giocatore parallelo presenta alcuni problemi:

• in condizioni di torneo (cioè con limitazione del tempo di ricerca) le varie istanze sono eterogenee in termini di forza di gioco e ciò comporta un contributo negativo per il giocatore parallelo da parte

dei giocatori più deboli. I risultati sperimentali, infatti, hanno dimostrato che il giocatore parallelo è decisamente più debole di quello sequenziale più forte (fra quelli che lo compongono).

• i giocatori commerciali sono stati impiegati come delle scatole nere, cioè senza conoscerne l'algoritmo e le euristiche di ricerca nonché la conoscenza catturata dalla funzione di valutazione. La conoscenza di queste informazioni avrebbe potuto aiutare nella definizione di un criterio di selezione più efficace nel migliorare la qualità di gioco del giocatore parallelo.

Intuitivamente il successo di questo nuovo approccio dipende dalla scelta dell'insieme di istanze di ricerca: una scelta casuale (come quella descritta in [Alt91]) non appare certo il metodo più efficace.

Non meno importante, però, è la determinazione di un criterio di selezione che rifletta la conoscenza delle caratteristiche delle istanze e sia quindi in grado di "pesare" la qualità della ricerca di ciascuna istanza.

Le finalità che si intende perseguire con gli esperimenti che verranno descritti sono:

• stabilire quali combinazioni di istanze inducono la migliore qualità di gioco. L'intento non è di realizzare un unico e fortissimo giocatore parallelo, bensì classificare in funzione della loro efficacia le varie possibilità di associare istanze di ricerca diverse. Una combinazione di istanze verrà indicata efficace solo se il giocatore parallelo che ne scaturisce è più forte di tutte le sue istanze considerate singolarmente.

• individuare criteri di selezione generali e possibilmente indipendenti dal dominio capaci di combinare al meglio le qualità delle istanze di ricerca.

La grossa differenza rispetto al lavoro in [Alt91] sarà il progetto delle istanze. In particolare si tenterà di creare classi di istanze, dove ci si aspetta che i componenti di ciascuna classe "stiano bene insieme", vale a dire abbiano qualche caratteristica comune che faccia ben sperare nella loro combinazione all'interno dello stesso giocatore parallelo.

Se questo tipo di esperimenti avesse successo, cioè si riuscisse a stabilire uno o più metodi di combinare insieme più istanze di ricerca tali che il giocatore risultante sia più forte di ogni singola istanza, questo risultato aprirebbe nuove prospettive di ricerca; con le argomentazioni che seguiranno verrà chiarito il perché di questa affermazione.

Una naturale obiezione a questo tipo di approccio è la notevole ridondanza (dovuta alla visita multipla dello stesso albero di gioco) e lo "spreco" di risorse: il lavoro di un'istanza può risultare inutile qualora la mossa da essa proposta non sia la prescelta dal criterio di selezione.

Tuttavia allo stato dell'arte, e come abbiamo visto nel Capitolo 5, esiste un limite per il numero di processori impiegati in un algoritmo parallelo di ricerca oltre il quale non si ottiene alcun beneficio

dall’aumento di risorse di elaborazione. Una soluzione a tale limitazione può essere quella di combinare le due forme di parallelismo: ogni istanza di ricerca è costituita da un pool di processori che realizzano la ricerca dell’albero di gioco in accordo ad un certo algoritmo parallelo (parallelismo introdotto a basso livello); a più alto livello, invece, sta il giocatore parallelo oggetto della presente trattazione, il quale non ha alcuna visibilità del parallelismo interno delle istanze. Il vantaggio che ne deriva è evidente: ogni istanza ha migliorato la sua qualità di gioco (la ricerca parallela consente una esplorazione più in profondità dell’albero) e di ciò risente positivamente la qualità del giocatore complessivo. Sarebbe di grosso interesse quantificare sperimentalmente questo miglioramento verificando così anche le caratteristiche ed i problemi di questa soluzione mista.

Alla luce di quanto detto la presenza di ridondanza va accolta come un fatto positivo in quanto offre un ampio margine di miglioramento per i futuri perfezionamenti del metodo.

Il presente lavoro intende indagare, in particolare, metodologie di distribuzione della conoscenza terminale. In particolare nel prossimo paragrafo sarà discusso il legame fra conoscenza e teoria dei giochi e presentato uno studio della conoscenza terminale nel gioco degli scacchi. Quest’ultimo costituirà un utile fondamento al seguente progetto e realizzazione di un giocatore parallelo costituito da istanze con diversa conoscenza terminale.

La distribuzione della conoscenza dirigente sarà tuttavia oggetto di una breve discussione in cui saranno motivate alcune intuizioni sulla reale efficacia di questo approccio e proposte alcune linee di ricerca futura.

6.2 Conoscenza e dominio dei giochi

Con la crescente consapevolezza delle potenzialità raggiunte dai sistemi esperti, l’ingegneria della conoscenza è divenuta una disciplina unanimemente riconosciuta. La responsabilità principale dell’ingegnere della conoscenza è raccogliere l’esperienza acquisita dagli esperti di un certo dominio ed esprimerla in una forma manipolabile da un calcolatore.

Per la maggior parte dei domini esiste un’incredibile quantità di informazioni disponibili, non tutte necessarie per ottenere prestazioni di alto livello. In generale un programma di gioco non può conoscere qualsiasi informazione necessaria ad ottenere una perfetta condotta di gioco: quale parte della conoscenza è allora importante?

Ad ogni porzione di conoscenza è associata un’importanza relativa. Ad esempio nel gioco degli scacchi conoscere che "il vantaggio di una regina di solito implica vincere la partita" appare molto importante. Al contrario, un giocatore professionista di scacchi potrebbe non incontrare in tutta la sua vita un finale di partita particolare; si potrebbe concludere che conoscere come giocare correttamente quel certo finale è relativamente poco importante:

tale però rimane solo fino a quando quel finale non occorre realmente!

Comprendere l’importanza relativa delle "porzioni" di conoscenza è un passo necessario per lo sviluppo di un programma di gioco che massimizzi la qualità delle sue scelte minimizzando però ridondanza ed inefficienza.

Riguardo il gioco degli scacchi è stata accumulata, nei secoli, un’enorme quantità di conoscenza. A partire da questa l’ingegnere della conoscenza deve determinare quale sottoinsieme è più importante sia contenuto in un programma; nozioni meno importanti potranno tuttavia essere aggiunte in seguito. Il consulto con altri esperti può probabilmente essere necessario per un consenso sulle scelte di base; dopo questo, però, la ricerca dovrà fare affidamento unicamente su tecniche trial-and-error e sull’esperienza via via acquisita.

È convinzione diffusa che per aumentare la forza di un giocatore artificiale è sufficiente arricchirlo di quanta più conoscenza è possibile. Essa ha fondamento nella maggioranza dei sistemi esperti (o almeno appare averne); in un programma di gioco, tuttavia, tale intuizione non ha valore assoluto. Si osservi che le componenti della conoscenza devono operare come una squadra e quindi la qualità di gioco dipende anche dagli effetti di una loro combinazione; potrebbe quindi accadere che risulti migliore non un programma con maggiore conoscenza, ma uno le cui componenti operano meglio insieme [Sch86].

Argomento della presente sezione di lavoro è il progetto di un giocatore parallelo costituito da istanze di ricerca differenti; ciò che diversifica tali istanze è la rispettiva conoscenza del dominio di applicazione. Questo tipo di problema introduce un livello di complessità supplementare: all’individuazione della conoscenza si aggiunge la questione della sua distribuzione. L’obbiettivo che si intende perseguire è lo sviluppo di metodologie (possibilmente indipendenti dal dominio) riguardanti questo secondo aspetto. La sperimentazione di queste non può però prescindere

da argomenti che riguardano il dominio cui sono applicate: come superare questo ostacolo?

La soluzione è stata quella di utilizzare i risultati noti di uno studio molto articolato di principi di selezione ed ordinamento della conoscenza applicati al dominio degli scacchi [Sch86]. Tali risultati riguardano proprio il dominio di sperimentazione del presente lavoro; essi costituiranno quindi la risposta a qualsiasi interrogativo che, nel corso del progetto e sperimentazione della distribuzione della conoscenza, sarà inerente il particolare dominio cui la distribuzione è applicata. Gli argomenti della citata ricerca sono ora presentati in sintesi.

6.2.1 Uno studio della conoscenza terminale nel gioco degli scacchi

Il lavoro di Schaeffer costituisce un'esplorazione sistematica delle componenti di un giocatore sequenziale di scacchi; particolare attenzione è dedicata allo studio della conoscenza terminale, cioè quella incorporata nella funzione di valutazione [Sch86].

Tale tipo di conoscenza non è basata su regole; essa consiste invece di frammenti di programma aventi lo scopo di riconoscere particolari caratteristiche e proprietà di una posizione del gioco e, per ciascuna di esse, indicare un valore numerico che ne rifletta il gradimento o meno da parte del giocatore cui spetta la mossa[27]. I valori così ottenuti sono combinati linearmente per determinare la valutazione complessiva della posizione.

In particolare sono state fissate da Schaeffer 8 categorie di conoscenza applicata al mediogioco di una partita di scacchi e di seguito valutata l'importanza relativa di ognuna. Tali categorie sono tipicamente rappresentate in ogni programma di scacchi e quindi la loro valutazione assume un significato generale. Esse sono elencate di seguito:

- materiale (M)
- spazio e mobilità (SM)
- debolezza pedonale (PW)
- sicurezza del re (KS)
- controllo del centro (CC)
- struttura pedonale (PS)
- punteggio incrementale (IS) indicante la somma di punteggi accumulati lungo il cammino dalla radice al nodo terminale assegnati in base a certe proprietà delle mosse incontrate
- pianificatore (PL) che assegna punteggi addizionali a mosse che rispettano un certo piano di azione

Il primo esperimento eseguito da Schaeffer è stato quello di aggiungere in modo incrementale le categorie di conoscenza e valutare la forza del giocatore così ottenuto. In Tab. 6.1 sono indicati i punteggi così ottenuti espressi nello standard

ELO di valutazione di giocatori di scacchi [CFC84]. Essi sono solamente una frazione dei giocatori valutati in [Sch86] e che possono essere ottenuti permutando l'ordine di inserimento delle categorie di conoscenza.

Programma	Punteggio
M	1110
M+SM	1404
M+SM+CC	1512
M+SM+CC+PW	1570
M+SM+CC+PW+IS	1556
M+SM+CC+PW+IS+KS	1750
M+SM+CC+PW+IS+KS+PS	1758
M+SM+CC+PW+IS+KS+PS+PL	1786

Tab. 6.1 Esperimento di Schaeffer di incremento della conoscenza

L'ordine con cui le categorie sono aggiunte in Tab. 6.1 è quello che ordina le stesse per importanza relativa (dedotto in base ai risultati di tutti gli esperimenti).

Le indicazioni principali che devono essere tratte da questi risultati sono:

• la più importante euristica di conoscenza, dopo naturalmente il conteggio del materiale M, è l'analisi dello spazio e della mobilità SM: lo dimostra l'incremento nel punteggio del giocatore determinato dall'introduzione nel giocatore di questa categoria di conoscenza.

• dopo l'inserimento di SM l'aggiunta di nuove categorie contribuisce ad un minimo guadagno nella forza del giocatore risultante; l'inclusione di IS pare addirittura avere effetti negativi. Questo è un esempio di conoscenza che offre beneficio solo se combinata con altre. È stato infatti verificato che le categorie KS, PS e PL hanno tutte effetti negativi sulle prestazioni se considerate singolarmente, ma in combinazione con IS il loro apporto diviene positivo.

• la sicurezza del re è una categoria molto particolare: nella maggior parte delle posizioni essa non ha importanza e in molte partite può avere un'influenza minima sul gioco; tuttavia in posizioni in cui essa ha peso, la sua presenza o assenza può fare la differenza fra una vittoria ed una sconfitta!

Esperimenti analoghi sono stati eseguiti partendo dal giocatore contenente tutte le categorie di conoscenza e valutando quelli ottenuti rimuovendo selettivamente una sola di esse. I risultati hanno rivelato che se SM è la prima porzione di conoscenza che deve essere aggiunta ad un programma, PW è quella che ha maggior senso non rimuovere[28]. Questo risultato è giustificato dal fatto che PW

gioca un ruolo molto importante in ambienti ricchi di conoscenza, mentre la sua efficacia relativa è molto minore in presenza di poca conoscenza. Ciò spiega come certa conoscenza necessiti del giusto ambiente con cui interagire per poter ottenere i migliori risultati.

6.3 La distribuzione della conoscenza terminale

Questa sezione del lavoro intende indagare il progetto e la realizzazione di una classe di giocatori paralleli costituiti da istanze di ricerca caratterizzate da funzioni di valutazione diverse (conoscenza terminale). Al fine di isolare lo studio della distribuzione di questa sola forma di conoscenza, la conoscenza dirigente, cioè l'unione di algoritmo ed euristiche di ricerca, sarà la stessa per tutte le istanze.

Ciò che si vuole ottenere sono indicazioni generali sull'efficacia di questo nuovo approccio; è dunque necessario selezionare funzioni di valutazione significative, che siano cioè il risultato di suggerimenti di giocatori esperti. La scelta di funzioni che originino giocatori molto deboli o comunque dalle qualità di gioco estremamente eterogenee potrebbe falsare la verifica sperimentale delle idee che saranno proposte. La scelta delle funzioni di valutazione non deve dunque essere casuale.

In letteratura sono riportate numerose raccolte di funzioni di valutazione proposte da esperti: una possibile soluzione è quella di implementare fedelmente alcune di esse. Questo approccio presenta alcuni problemi:

• l'implementazione exnovo di funzioni di valutazione può originare bug di programmazione oppure non essere la più efficiente per quella funzione. Questo secondo aspetto appare molto interessante; si consideri ad esempio il progetto di una routine per il calcolo della mobilità dei pezzi. Questa conoscenza è molto importante (cfr. 6.2.1); tuttavia il suo costo computazionale è elevato poiché sottintende il conteggio delle mosse legali per ogni pezzo: abbiamo verificato che l'implementazione di questa euristica di conoscenza come routine separata ha effetti disastrosi sulle prestazioni. La soluzione efficiente è invece integrare il suddetto conteggio all'interno del generatore di mosse.

L'implementazione efficiente delle funzioni di valutazione non è dunque immediata e richiede, invece, uno studio approfondito delle sue caratteristiche e della sua integrazione nella struttura di ricerca: l'analisi di queste attività esula dagli obbiettivi del presente lavoro.

• l'implementazione di funzioni di valutazione è operazione costosa e "rischiosa"; ciò è in contrasto con il desiderio di disporre di un numero elevato di esse, tale da definire un numero di giocatori paralleli sufficiente a dare valore statistico alla valutazione sperimentale della nuova filosofia di distribuzione.

I problemi discussi sono risolti facilmente attraverso la "rielaborazione" dell'implementazione di un'unica funzione di valutazione sufficientemente testata nella sua correttezza e qualità;

come è possibile ricavare da un'unica funzione una molteplicità indefinita di funzioni diverse?

Si consideri un giocatore artificiale P preesistente (ad esempio GnuChess). L'idea è quella di decomporre la funzione di valutazione di P nelle componenti (chiamate parametri della funzione) sommate linearmente[29] nella valutazione statica di un nodo terminale. Questa decomposizione equivale ad una separazione delle categorie di conoscenza terminale. Combinando insieme solo una parte della conoscenza contenuta nella funzione originaria si ottengono funzioni di valutazione completamente differenti.

Più formalmente, data la funzione di valutazione F sia D_f l'insieme delle categorie di conoscenza che scaturiscono dalla sua decomposizione: ciascun sottoinsieme di D_f individua una particolare funzione di valutazione.

Il numero di funzioni che possono essere dunque definite è pari alla cardinalità dell'insieme delle parti di D_f ($=2^N$ se N sono gli elementi di D_f). Tali considerazioni sono valide nell'ipotesi che ciascuna categoria mantenga, nella ricombinazione lineare, lo stesso peso che essa aveva nella funzione F originaria; in caso contrario il numero di possibili funzioni ottenibili attraverso il metodo descritto è infinito perché tale è il numero di possibili attribuzioni di pesi.

Le istanze di ricerca che costituiranno il generico giocatore parallelo saranno caratterizzate da una diversa funzione di valutazione scelta fra quelle ottenute nel modo descritto.

Il risultato di questo approccio è avere integrato istanze di ricerca con diversa conoscenza terminale. In particolare esse eseguono una valutazione specializzata di una posizione in quanto la loro conoscenza terminale è una parte di quella generalmente contenuta in un programma: ad esempio avremo un'istanza che valuta soltanto la mobilità dei pezzi, un'altra il materiale ed un'altra ancora la struttura pedonale e la sicurezza del re.

Intuitivamente ciascuna istanza così ottenuta è più debole del programma originario in quanto ha minore conoscenza rispetto ad esso. È lecito dunque chiedere quali benefici possa portare la forma di parallelismo in analisi; le seguenti osservazioni sorreggono l'ipotesi che questo metodo migliori realmente la qualità di gioco del giocatore iniziale:

• nell'ipotesi che ogni categoria di conoscenza del giocatore di partenza P sia contenuta in almeno un'istanza, complessivamente il giocatore parallelo dispone della stessa conoscenza di P;

• la classica ricerca su albero di gioco condotta con il metodo $\alpha\beta$ consente di individuare la migliore mossa, ma non permette di dedurre una stima del valore delle mosse restanti. Il metodo in oggetto, invece, consente di ricavare un insieme di mosse valide, risultate le migliori rispetto a diversi tipi di valutazione delle posizioni terminali;

• la funzione di valutazione di ogni singola istanza è più semplice di quella originale e quindi il suo calcolo più efficiente. Quello che si attende è pertanto un guadagno in termini di profondità di ricerca (seppure minimo).

È dimostrato da studi psicologici che la scelta della mossa da parte di un giocatore esperto si riduca ad un'analisi di poche mosse significative [HLMSW92]. In generale tali mosse "valide" presentano vantaggi diversi l'una rispetto all'altra. Quello che ci si aspetta dal giocatore parallelo in analisi è che le varie istanze propongano mosse diverse, ciascuna migliore secondo una particolare visuale di gioco.

Un evidente problema di questo metodo è che talvolta mosse ovviamente svantaggiose non siano riconosciute come tali dalle funzioni di valutazione parziale e quindi possano non essere rifiutate immediatamente dall'algoritmo $\alpha\beta$. A parziale soluzione di questo problema può aiutare l'osservazione che qualsiasi tipo di valutazione non può prescindere da certi tipi fondamentali di analisi, quale ad esempio quella del materiale. Un vincolo che deve essere dunque imposto è che la categoria di conoscenza che valuta il materiale sia presente nella funzione di valutazione di tutte le istanze.

6.3.1 La struttura del giocatore parallelo

Il linguaggio Linda ed il programma di scacchi GnuChess (Capitolo 3) costituiscono gli strumenti attraverso i quali sarà descritto e sperimentato un esempio concreto di giocatore parallelo basato sul concetto di distribuzione della conoscenza terminale. Il ruolo di questi strumenti è il seguente: a partire da un unico giocatore sequenziale (GnuChess) si vuole creare (attraverso le primitive Linda) una struttura di istanze di questo giocatore che cooperano attraverso lo spazio delle tuple e che siano parametriche rispetto alla funzione di valutazione (Fig. 6.1a).

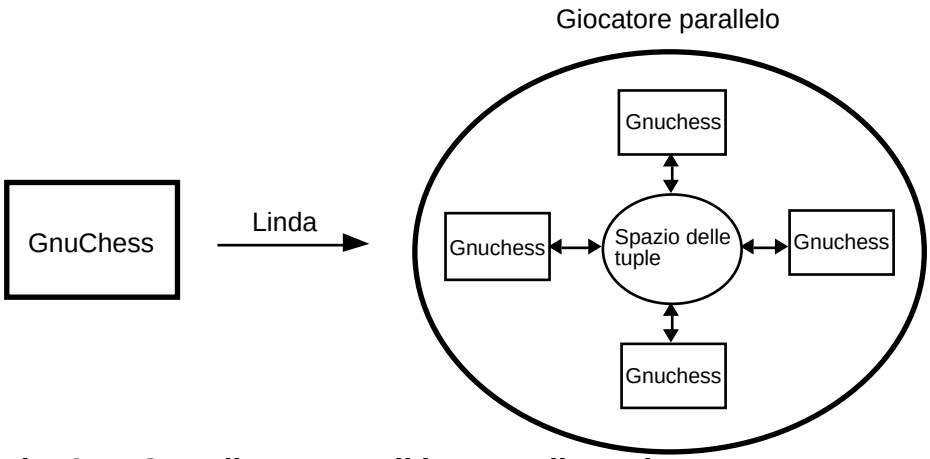


Fig. 6.1a Coordinamento di istanze di un giocatore sequenziale

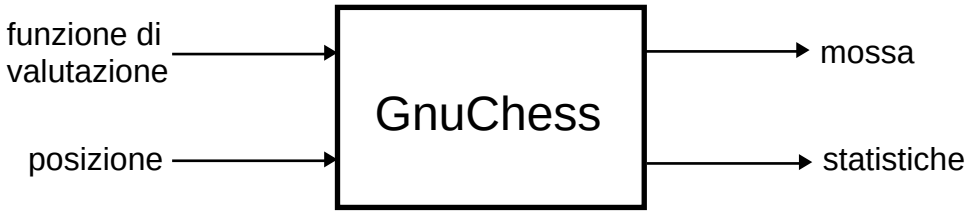


Fig. 6.1b Interfaccia del giocatore sequenziale

Questo tipo di progetto intende verificare, fra l'altro, l'efficacia di Linda nella "parallelizzazione" di programmi sequenziali; con questo termine si intende il "riutilizzo" di moduli di programmi sequenziali nel definire programmi paralleli dove essi descrivono il flusso di controllo interno dei processi. Questo tipo di approccio alla programmazione parallela è indubbiamente vantaggioso in quanto riduce questa attività alla sola descrizione del coordinamento inter-modulo, esonerando il programmatore dalla definizione della parte sequenziale dei processi. La sua applicabilità non sempre è possibile poiché fortemente influenzata dalla portabilità e dalla modularità del software sequenziale nonché dalle caratteristiche del linguaggio di coordinamento.

La funzionalità di ricerca di GnuChess è stata quindi riutilizzata nella sua interezza, considerando essa un blocco unico, una scatola nera di cui interessa la sola interfaccia (Fig. 6.1b). Le uniche modifiche ad essa apportate hanno riguardato la funzione di valutazione nei riguardi della quale è stata resa parametrica; questa operazione ha richiesto un cambiamento minimo del codice sequenziale poiché, come spiegato in 6.3, la funzione originaria deve essere reimpiegata integralmente, anche se decomposta fisicamente nelle porzioni di codice che calcolano euristiche di valutazione distinte.

In 2.4.2.3.4 è stata proposta una possibile decomposizione logica della funzione di valutazione di GnuChess: essa costituisce il riferimento per l'applicazione del metodo di generazione di funzioni di valutazione descritto in 6.3. Di seguito sono elencate le categorie di conoscenza individuate dalla suddetta decomposizione:

- materiale (M)
- valore posizionale dei pezzi (B)
- mobilità (di torre e alfiere) e combinazioni di attacco (X)
- sicurezza del re (K)
- struttura pedonale (P)
- protezione dei pezzi (A)
- relazione pezzi-struttura pedonale (R)

La scelta di decomporre la funzione di valutazione secondo le categorie di conoscenza appena elencate non è stata casuale, ma guidata dalle indicazioni del lavoro di Schaeffer

[Sch86]: si osservi, infatti, la corrispondenza fra molte di queste euristiche e quelle utilizzate nel suo esperimento e riportate in 6.2.1.

La codifica della funzione di valutazione che comprende le euristiche:

- materiale (M)
- sicurezza del re (K)
- struttura pedonale (P)

è:

M	B	X	K	P	A	R
1	0	0	1	1	0	0

$\longrightarrow (1001100)_2 = (76)_{10}$

Fig. 6.2 Codifica numerica della funzione di valutazione

La funzione di valutazione di una generica istanza è ottenuta includendo o meno il contributo che deriva dal calcolo di ciascuna delle N (=7) euristiche di valutazione elencate. Le funzioni che sono originate da questo metodo possono essere facilmente codificate con un intero appartenente all'intervallo $[0,2^N-1]$; in Fig. 6.2 è descritto un esempio di tale codifica.

Il flusso di controllo che scandisce il calcolo della funzione di valutazione è quindi invariato rispetto a quello di GnuChess; la sola eccezione è l'inserzione di controlli dinamici che verificano, in base alla codifica della funzione, quali routine di valutazione debbano essere attraversate.

Quanto descritto permette di disporre di un codice unico per l'implementazione di tutte le funzioni di valutazione essendo la specializzazione di quest'ultime rimandata al tempo di esecuzione.

Oltre all'evidente vantaggio di non dover programmare ogni diversa funzione di valutazione, la struttura descritta si rivela estremamente flessibile: si osservi come sia possibile cambiare dinamicamente la funzione di valutazione corrente semplicemente modificandone la codifica numerica. Il prezzo di questi benefici è il costo dei controlli dinamici che implementano la decodifica della funzione di valutazione; in generale esso può essere considerato trascurabile se confrontato con il tempo di esecuzione complessivo del programma[30].

Quando e come avviene la cooperazione fra le istanze di GnuChess che compongono il giocatore parallelo?

La ricerca di ciascuna istanza è sequenziale ed indipendente dalle altre: durante il suo svolgimento non si avrà dunque alcuna interazione fra processi. La cooperazione è stabilita nel momento in cui, terminata la rispettiva ricerca, le istanze

si "incontrano" per suggerire la propria mossa e "decidere" (applicando un criterio di selezione) quale fra quelle proposte sia preferibile giocare.

Il modello di programmazione parallela con cui è stato implementato il coordinamento delle attività descritte è il master-worker. In particolare una delle istanze (master) è stata arricchita con funzionalità di gestione e coordinamento dell'operato delle altre (worker).

Il flusso di controllo del processo coordinatore scandisce gli stadi attraversati dal giocatore parallelo:

• la creazione dei processi worker: uno dei parametri del processo coordinatore è la descrizione del giocatore parallelo intesa come la specifica del numero P di istanze che lo compongono e della funzione di valutazione (codificata) di ciascuna di esse.

Sulla base di queste informazioni sono create (P-1) istanze di ricerca cui è comunicata la rispettiva funzione di valutazione; anche il processo coordinatore sarà impegnato nella ricerca e quindi si attribuirà esso stesso una delle funzioni di valutazione previste.

La creazione della struttura di istanze avviene una sola volta ed è concentrata nella fase preliminare del gioco.

• la ricerca: il processo coordinatore implementa l'interfaccia di i/o del giocatore parallelo. In particolare esso è l'unica istanza cosciente del momento in cui il giocatore parallelo deve eseguire la scelta della mossa: quando questa è necessaria esso richiede a ciascuna istanza che sia iniziata la rispettiva ricerca. Di seguito anch'esso intraprende la personale ricerca in parallelo alle altre.

Non è necessario che la descrizione del lavoro richiesto ai processi worker contenga la rappresentazione completa dello stato del gioco (disposizione dei pezzi, possibilità di arrocco, ecc.) poiché ogni istanza ne mantiene una copia locale; l'aggiornamento di questa informazione richiede però che il processo master comunichi ai worker le mosse che vengono via via realmente eseguite. È preferita questa scelta perché il costo della gestione locale dello stato del gioco è decisamente inferiore al suo periodico trasferimento totale.

La durata della ricerca è la stessa per tutte le istanze ed è costante durante tutto l'arco di una partita. La terminazione delle ricerche sarà quindi pressoché simultanea: ciò ha il vantaggio di impegnare produttivamente tutte le istanze durante la fase complessiva di ricerca.

L'implementazione di euristiche di amministrazione del tempo è stata evitata perché dipendenti dal dominio e di difficile concezione in un giocatore parallelo come quello in analisi: le istanze sono indipendenti e quindi il risparmio di

tempo deciso autonomamente da una potrebbe essere vanificato dalla ricerca prolungata di un’altra.

• l’applicazione del criterio di selezione: al termine della rispettiva ricerca ogni worker comunica al master la mossa ritenuta migliore. Essa è accompagnata da un insieme di dati statistici inerenti alcune proprietà della visita completata. Sulla base di queste informazioni e di quelle ottenute dalla propria ricerca il coordinatore applica il criterio di selezione determinando così la mossa definitiva. L’esito di questa scelta sarà comunque reso noto ai processi worker i quali potranno così aggiornare la copia locale dello stato del gioco.

In Fig. 6.3 è descritta in Linda l’architettura software appena discussa.

Si osservi che l’implementazione proposta non rispetta fedelmente il modello master-worker poiché in realtà i processi worker non sono indistinguibili, ma è attribuito loro un identificatore numerico al momento della rispettiva creazione.

La presenza di tale identificatore è giustificata dal fatto che il processo master deve conoscere non solo la mossa scelta da ciascun worker, ma anche da quale di questi essa è stata proposta.

```
void master (int n_istances,int *functions)
{
int my_function,quit,istance;
move *moves,move_selected,mv;
statistics *search_data,stats;
my_function=functions[0]; /* è assegnata la funzione di
valutazione del master */
/* segue la creazione dei worker e l’attribuzione delle
rispettive funzioni di valutazione */
for (istance=1;istance<n_istances;istance++)
       eval ("istance",worker(istance,functions[istance]));
NewGame (); /* è la funzione di GnuChess che inizializza lo
stato del gioco */
/* l’area di memoria per i vettori moves e search data deve
essere allocata dinamicamente */
memory_alloc (n_istances,&moves,&search_data);
quit=false;
while (quit)
       {
       if (side_to_move()==PARALLEL_PLAYER) /* la mossa
spetta al giocatore parallelo? */
              {
              /* è richiesta la ricerca di tutti i worker */
              for (i=1;i<=n_istances;i++)
                     out ("job",i,DO_SEARCH);
              /* anche il master conduce la sua esplorazione
dell’albero di gioco */
              moves[0]=SelectMove (my_function,
&search_data[0])
              /* inizia la raccolta delle mosse suggerite
dai worker */
              for (istance=1;istance<=n_istances;istance++)
                     {
```

```
                        in
("istance_move",?istance,?mv,?stats);
                        moves[istance]=mv;
                        search_data[istance]=stats;
                        }
move_selected=criteria(move,search_data); /*  criterio di
selezione */
                }
        else  /* deve essere raccolta la mossa
dell'avversario del giocatore parallelo */
                move_selected=other_player_search ();
        MakeMove (move_selected); /* aggiornamento dello
stato del gioco */
        if (quit=end_gamep())  /* fine partita? */
                for (istance=1;istance<n_istances;istance++)
                        out ("job",istance,QUIT);
        else  /* i worker devono  aggiornare la loro copia
dello stato del gioco */
                {
                out ("move_selected",move_selected);
                out ("sincr",n_istances-1);
                for (istance=1;istance<n_istances;istance++)
                        out ("job",istance,DO_MOVE);
                in ("sincr,0); /* per problemi di correttezza
il master non può inserire nuovi job
in agenda prima che tutti i worker abbiano raccolto la
richiesta di aggiornamento della copia dello stato del
gioco */
                inp ("move_selected",move_selected);
                }
        }
for (i=1;i<n_istances;i++) /* è verificata la terminazione
corretta dei worker */
        in ("istance",0);
return ();
}
int worker (int identifier,int my_function)
{
int job,quit,s;
move mv;
statistics stats;
NewGame ();
quit=false;
while (!quit)
        {
        in ("job",identifier,?job); /* ricezione job */
        switch (job);
                {
                case DO_SEARCH:
                        /* è richiesta la ricerca sequenziale
della posizione corrente */
                        mv=SelectMove (my_function, &stats)
                        out
("istance_move",identifier,mv,stats);
                        break;
                case DO_MOVE:
                        rd ("move_selected",?mv);
                        MakeMove (mv); /* aggiornamento dello
stato del gioco */
                        in ("sincr",?s);
                        out ("sincr",s-1);
                        break;
                case QUIT:
                        quit=true;
                        break;
                default:
                        break;
                }
        }
return (0);
```

```
}
```

Fig. 6.3 Coordinamento delle istanze di ricerca

L’attribuzione di un nome ai processi worker consente inoltre di risolvere in modo efficiente problemi di correttezza legati al prelievo dei lavori. A riguardo si osservi che la forma di comunicazione che deve essere stabilita fra il coordinatore e le altre istanze è asimmetrica in uscita con diffusione: l’esecuzione di un lavoro è richiesta a tutte le istanze. Al contrario, la forma di comunicazione fra il master e i worker stabilita dal paradigma omonimo è anch’essa asimmetrica in uscita, ma non avviene per diffusione poiché ogni lavoro deve essere eseguito da un unico worker.

6.3.2 La determinazione finale della mossa: il criterio di selezione

Al momento di scegliere la migliore mossa in una data posizione il giocatore parallelo origina tante ricerche quante sono le istanze che lo compongono. Ogni ricerca è indipendente dalle altre e viene condotta sullo stesso albero di gioco; i nodi visitati differiranno da un’istanza all’altra poiché i diversi valori attribuiti ai nodi terminali dalle rispettive funzioni di valutazione inducono tagli dell’albero discordanti.

Ogni istanza di ricerca ritorna la mossa risultata migliore in seguito alla propria visita ed il valore minimax associato. Il problema della determinazione della mossa che il giocatore parallelo dovrà giocare si riduce ora alla selezione di una fra tutte le mosse proposte dalle istanze di ricerca. Tale scelta è formulata in accordo ad un certo criterio chiamato appunto di selezione. La discussione seguente intende presentare una rassegna di criteri di selezione ponendo maggiore enfasi su quelli indipendenti dal dominio di applicazione.

6.3.2.1 Criteri di selezione a maggioranza semplice e generalizzata (con pesi costanti)

Il criterio di selezione più ovvio è quello detto a maggioranza semplice (o democratico): la mossa che ha ottenuto maggiori preferenze è quella che sarà effettivamente giocata; in caso di parità la scelta è casuale.

Il criterio democratico è indipendente dal dominio ed è quindi di applicazione generale. Tuttavia esso soffre, a causa della sua semplicità, di alcuni problemi:

- non viene tenuta in considerazione la forza relativa delle istanze;
- la mossa selezionata è (in generale) il risultato di un’analisi parziale delle sue proprietà (a causa della specializzazione delle funzioni di valutazione). La

situazione sgradevole che può quindi scaturire è che tale mossa, risultata la migliore sotto molti punti di vista, risulti scadente sotto alcuni altri pregiudicando drammaticamente la qualità del gioco qualora venisse scelta; il criterio democratico non fornisce strumenti per rilevare e risolvere questo problema. Sarebbe importante stimare con quale frequenza può presentarsi questo fenomeno ed al suo occorrere quale influenza esso può avere sulla forza del giocatore artificiale.

• vi sono aspetti di una posizione del gioco che hanno maggiore importanza. È indubbio che negli scacchi un vantaggio di materiale conduce quasi certamente alla vittoria, mentre lo stesso non è così evidente per un vantaggio nella struttura pedonale o nel controllo del centro.

Gli esperimenti presentati in 6.2.1 hanno dimostrato che alcune euristiche di valutazione sono più efficaci di altre nel migliorare la forza di gioco. Va inoltre considerato come l'importanza di un'euristica sia dipendente dalla posizione corrente del gioco e pertanto l'importanza relativa delle euristiche di valutazione può variare durante l'arco di una partita. Ad esempio l'analisi della sicurezza del re assume importanza quando la partita sta per volgere alla sua fase finale, mentre il controllo del centro ha maggiore peso negli stadi iniziale e centrale. Il criterio di selezione a maggioranza semplice non tiene conto di queste considerazioni; il rischio è dunque che in una situazione di gioco in cui non abbiano incidenza valutazioni riguardo la sicurezza del re o la struttura pedonale, una mossa venga erroneamente scelta solo perché risultata migliore sotto questi punti di osservazione.

Per risolvere (parzialmente) alcuni dei problemi presentati si può pensare di adottare una versione generalizzata del criterio di selezione a maggioranza.

L'idea è di associare un peso a ciascuna delle mosse proposte dalle istanze di ricerca. Questo peso riflette l'importanza attribuita ad una certa mossa per essere risultata migliore secondo un tipo di valutazione piuttosto che un altro. La somma dei pesi accumulati da una mossa scelta da una o più istanze ne determina il valore. La mossa selezionata è dunque quella cui è associato il maggior valore.

È evidente come il criterio di selezione a maggioranza semplice rappresenti un caso particolare della

versione generalizzata dove tutti i pesi hanno valore unitario.

Tale metodo permette di eliminare situazioni di scelta casuale se si stabiliscono pesi che impediscano a più mosse di accumulare lo stesso valore complessivo.

Il vantaggio che deriva da questo criterio è la possibilità di attribuire maggiore importanza ad euristiche di valutazione considerate più affidabili; di riflesso avranno peso superiore le scelte di quelle istanze la cui funzione di valutazione contiene le categorie di conoscenza terminale ritenute più efficaci.

Quando avviene l'attribuzione dei pesi? E sulla base di quali informazioni?

La risposta più semplice a tali quesiti è prevedere che i pesi siano costanti, cioè indipendenti dalla posizione in cui deve avvenire la scelta della mossa. Ciò significa che a tutte le mosse proposte da una stessa istanza verrà sempre attribuito lo stesso peso. In questo caso il peso diviene una proprietà costante dell'istanza, o meglio della sua funzione di valutazione: esso costituisce un indice generale dell'importanza relativa della conoscenza in essa contenuta.

Generalizzazione della soluzione con pesi costanti è il criterio con pesi variabili: i pesi attributi alle mosse scelte da un'istanza possono variare da una posizione di gioco all'altra. Questo approccio identifica una classe di criteri di selezione estremamente complessa che sarà indagata nel paragrafo 6.3.2.3. Per il momento sarà valutata l'efficacia di alcuni dei criteri sin qui illustrati attraverso una loro applicazione al giocatore parallelo descritto in 6.3.1.

6.3.2.2 La valutazione sperimentale del giocatore parallelo

Ciò che si intende sperimentare sono:

• alcuni modi di distribuire la conoscenza terminale fra le istanze e

• alcuni criteri di selezione.

Dalle possibili combinazioni di questi due aspetti e al variare del numero delle istanze traggono origine un elevato numero di giocatori paralleli.

Il problema è confrontare la qualità di gioco di tali giocatori eseguendo esperimenti di durata non eccessiva, ma statisticamente attendibili.

Il criterio più ovvio per la valutazione della forza di un giocatore è osservarlo alle prese con una partita completa.

Ciò che si vorrebbe ottenere è un punteggio che sintetizzi la sua qualità di gioco rispetto agli altri giocatori paralleli. La soluzione più corretta sarebbe quella di far giocare una serie di N partite fra ciascuna delle possibili coppie di giocatori paralleli; tuttavia il numero totale di partite risulterebbe eccessivo:

$\frac{N \cdot K \cdot (K-1)}{2}$ se K è il numero di giocatori

In alternativa a quella appena descritta, la soluzione adottata è stata quella di fissare un avversario unico per tutti i giocatori paralleli riducendo così il numero delle partite di valutazione a $N \cdot K$.

Il confronto fra la qualità dei giocatori paralleli è dunque indiretto poiché avviene comparando il numero di vittorie riportate da ciascuno nei riguardi dell'avversario comune.

Il giocatore scelto come avversario dei giocatori paralleli è proprio la versione originale di GnuChess[31]. Questa soluzione ha il pregio di evidenziare direttamente se la distribuzione della conoscenza ha portato benefici rispetto al giocatore originario (GnuChess) in cui la stessa conoscenza è invece concentrata.

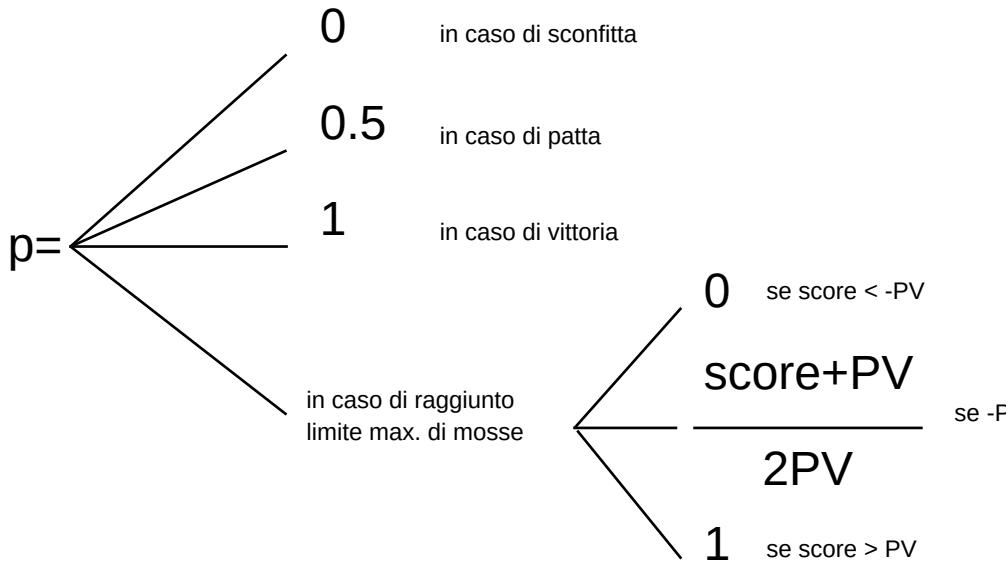
Sono stati inoltre ristretti i limiti di una generica partita escludendo la sua fase finale; il motivo è che durante questo stadio ha importanza solo una limitata porzione di conoscenza terminale e ciò potrebbe condizionare erroneamente la valutazione complessiva dei giocatori. In particolare la partita è troncata dopo un prefissato numero di mosse; essa può quindi concludersi per i seguenti motivi:

- situazione di patta
- situazione di matto
- raggiunto limite massimo di semi-mosse (=100)

È stato inoltre inibito l'utilizzo del libro di aperture poiché questo potrebbe condurre il gioco verso posizioni "non equilibrate", dove ad esempio una forte debolezza della struttura pedonale potrebbe favorire quei giocatori paralleli dove esiste una maggiore ridondanza fra le istanze di questa categoria di conoscenza.

In particolare ogni giocatore parallelo G disputa contro GnuChess un torneo di 20 partite; il colore di gioco è assegnato equamente ai due giocatori: G disputerà 10 partite con il bianco e 10 con il nero.

Il valore finale di G è calcolato sommando i punteggi ottenuti in ogni partita; il punteggio p ricavato da G in una partita dipende dalla condizione di terminazione di quest'ultima; i valori che esso può assumere sono riportati in Fig. 6.4.



PV = valore posizionale equivalente ad un pedone (100)
score = stima del valore posizionale della posizione finale dal punto di vista del giocatore parallelo

Fig. 6.4 Attribuzione del punteggio in una singola partita

Un problema molto difficoltoso è proposto dai casi di terminazione forzata della partita in cui non esiste un chiaro vincitore; in generale, tuttavia, uno dei giocatori avrà accumulato un vantaggio posizionale che a lungo termine potrebbe con molta probabilità rivelarsi decisivo: in questo caso esso viene ritenuto vincitore della partita.

Per tenere conto di queste situazioni viene utilizzata la funzione di valutazione di GnuChess (che contiene l'insieme completo di euristiche di valutazione) per stimare il vantaggio posizionale (eventualmente negativo) del giocatore parallelo G. Esso viene confrontato con un valore posizionale costante per stabilire la vittoria o meno di G; tale valore è fissato arbitrariamente pari al valore PV di un pedone. Cosa succede se $|score| \leq PV$?

Piuttosto che definire la fascia intermedia [-PV,+PV] come "area di patta" ed associare ad essa il punteggio 0.5 di equilibrio, si è preferito dare

significato al valore che score assume all'interno di tale intervallo ed assegnare un valore proporzionale ad esso (Fig. 6.5).

Dalla somma dei contributi di tutte le partite si ottiene il valore complessivo di G; il valore di equilibrio nei confronti di GnuChess per il punteggio finale è dunque 10.

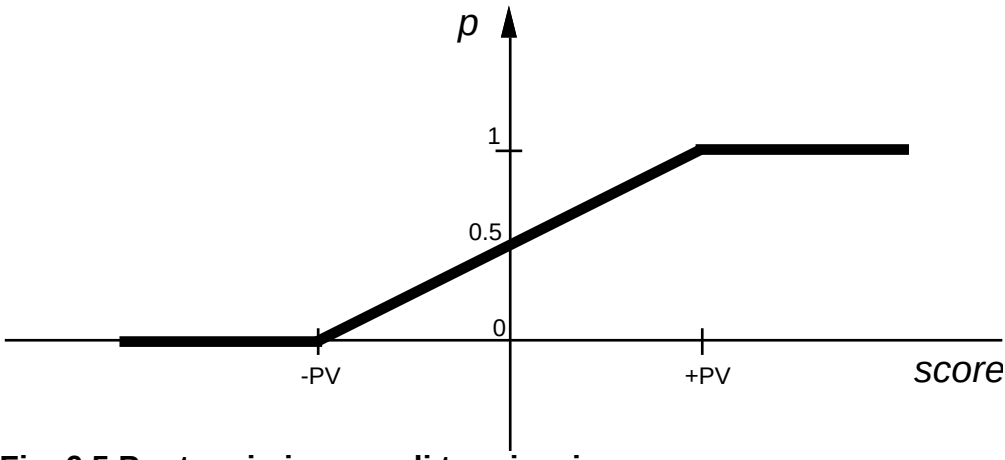


Fig. 6.5 Punteggio in caso di terminazione per raggiunto limite di mosse

L'ultimo parametro degli esperimenti che deve essere ancora specificato è la durata massima della ricerca di una singola istanza che è stata fissata a 30 secondi.

Deve inoltre essere sottolineato che le modalità di gioco di GnuChess saranno programmate in modo che siano disabilitate le funzionalità di amministrazione del tempo (ogni mossa impegnerà completamente i 30 secondi fissati) e di ricerca durante l'attesa della mossa dell'avversario (GnuChess rimarrà inoperoso durante questo periodo).

Chiarite le modalità con cui sarà eseguita la valutazione dei giocatori paralleli è ora necessario fissare l'identità di quelli che saranno effettivamente valutati.

È conveniente stabilire una notazione simbolica che permetta di descrivere sinteticamente le istanze che compongono un certo giocatore. Si osservi che a ciascuna delle euristiche di valutazione di GnuChess è stata associata una lettera dell'alfabeto (cfr. 6.3.1); tali lettere sono utilizzate per indicare l'insieme di conoscenza terminale contenuta in ciascuna delle istanze.

Ad esempio con la notazione (MB)(MX)(MBXK) si intende indicare un giocatore parallelo costituito da 3 istanze di ricerca, dove ciascuna utilizza nella propria funzione di valutazione le euristiche racchiuse dalla coppia di parentesi corrispondente.

La generazione dei giocatori paralleli che saranno valutati non è casuale, ma in accordo a criteri (intuitivi) di distribuzione della conoscenza: saranno adottati 4 di essi. L'applicazione di alcuni di questi necessita dell'ordinamento per importanza relativa delle euristiche di valutazione.

L'ordinamento delle euristiche di GnuChess è stato fissato facendo tesoro delle indicazioni fornite dagli esperimenti di Schaeffer (cfr. 6.2.1); esso è in sintesi il seguente: M, B, X, K, P, A, R.

La descrizione dei criteri di distribuzione è indipendente dal numero e dall'identità delle euristiche di valutazione: sia dunque $E_1,E_2,\ldots,E_N$ una generica sequenza ordinata di esse.

Sia inoltre n (numero di istanze) $\leq$ N (numero complessivo di euristiche); sono allora definiti i seguenti criteri di distribuzione della conoscenza:

• a. minima conoscenza: $G(n)=(E_1)(E_2)\ldots(E_n)$.
Ogni istanza contiene una sola euristica di valutazione fra le prime n dell'ordinamento.

• b. distribuzione bilanciata: le funzioni di valutazione di tutte le istanze contengono lo stesso numero di euristiche; inoltre ogni euristica è presente in uno stesso numero di istanze (compatibilmente con il rispetto del primo vincolo).

• c. distribuzione sbilanciata:
$G(n)=(E_{1\ldots}E_N)(E_1)\ldots(E_{n/2})(E_1E_{2\ldots}E_{N-1})\ldots(E_1E_2\ldots E_{n/2}E_{n/2+1}\ldots E_N)$.

Una delle istanze ha la stessa funzione di valutazione di GnuChess; metà delle altre istanze contiene solo una euristica fra le prime n/2 dell'ordinamento, mentre le restanti contengono tutte le euristiche eccettuato una fra le (n/2-1) meno efficaci.

• d. distribuzione incrementale:
G(n)=(E1)(E1E2)…(E1E2…En).

La prima istanza contiene la sola conoscenza più importante, la successiva anche la seconda dell'ordinamento, ecc.

Si osservi che i metodi b. e c. possono essere applicati anche se n>N; il metodo b., inoltre, non richiede un ordinamento preliminare delle categorie di conoscenza.

Ciascuno dei metodi elencati è stato applicato nella definizione di giocatori costituiti da un numero di istanze che va da 3 a 6. Sono dunque scaturiti 16 giocatori paralleli la cui natura è descritta in Tab. 6.2.

Si osservi che la conoscenza relativa al materiale è presente in tutte le istanze (per i motivi spiegati in 6.3): i criteri di distribuzione tengono quindi conto solamente delle restanti euristiche di valutazione.

Criterio di distribuzione	Etichetta	Giocatore parallelo
Minima conoscenza	3a	(MB)(MX)(MK)
	4a	(MB)(MX)(MK)(MP)
	5a	(MB)(MX)(MK)(MP)(MA)
	6a	(MB)(MX)(MK)(MP)(MA)(MR)
Distribuzione bilanciata	3b	(MBXP)(MBKR)(MXKA)
	4b	(MBXA)(MBKR)(MXPA)(MKPR)
	5b	(MBXK)(MBKA)(MBPR)(MXKR)(MXPA)
	6b	(MBXK)(MBKA)(MBPR)(MXPA)(MXAR)(MKPR)
Distribuzione sbilanciata	3c	(MB)(MX)(MBXKPAR)
	4c	(MB)(MX)(MBXKPAR)(MBXKPA)
	5c	(MB)(MX)(MK)(MBXKPAR)(MBXKPA)
	6c	(MB)(MX)(MK)(MBXKPAR)(MBXKPA)(MBXKPR)
Distribuzione incrementale	3d	(MB)(MBX)(MBXK)
	4d	(MB)(MBX)(MBXK)(MBXKP)
	5d	(MB)(MBX)(MBXK)(MBXKP)(MBXKPA)
	6d	(MB)(MBX)(MBXK)(MBXKP)(MBXKPA)(MBXKPAR)

Tab. 6.2 Giocatori paralleli con distribuzione della conoscenza terminale.

Quale criterio di selezione sarà adottato dai giocatori paralleli?

A riguardo sono stati stabiliti 2 differenti criteri del tipo a maggioranza con pesi costanti. Il peso che è attribuito ad una generica istanza è calcolato come segue:

• per ognuna delle euristiche di valutazione è fissato un peso che tenga conto dell'ordinamento indotto dalla loro importanza relativa; in Tab. 6.3 sono indicati i pesi attribuiti alle euristiche di GnuChess in relazione ai due criteri di selezione che si intende applicare.

	Criterio di selezione	
Euristica	Semplice	Generalizzato
M	130	30
B	121	21
X	115	15
K	110	10
P	106	6
A	103	3
R	101	1

Tab. 6.3 Attribuzione di pesi alle euristiche di valutazione

• sia $I=(Ei_1Ei_2\ldots Ei_k)$ una generica istanza e sia P: $E\rightarrow$int la funzione che associa ad un'euristica di valutazione il rispettivo peso.

Il peso dell'istanza I è calcolato come la media aritmetica dei pesi associati alle euristiche contenute nella sua funzione di valutazione:

$$peso(I) = \frac{\sum_{j=1}^{k} P(E_{ij})}{k}$$

• sia $M=\{m_1,\ldots,m_n\}$ l'insieme di mosse proposte dalle istanze di ricerca ed in particolare sia $I_i=\{I_{i1},\ldots,I_{ip}\}$ l'insieme di istanze che hanno "votato" la mossa mi (i=1..n); il valore v_i di tale mossa è calcolato come la somma dei pesi delle istanze contenute in I_i:

$$v_i = \sum_{j=1}^{p} peso(I_{ij})$$

Può finalmente avvenire la selezione finale:

mossa selezionata = $\max(v_1,\ldots,v_n)$.

Si osservi che i pesi fissati dal primo criterio in Tab. 6.3 fanno sì che il criterio a maggioranza sia equivalente alla sua versione semplificata: è scelta la mossa più votata; l'unica differenza è che la soluzione di situazioni di parità non è casuale, ma tiene conto dei pesi delle istanze.

Il criterio denominato "Generalizzato" è invece un classico esempio di criterio con maggioranza generalizzata; esso infatti prevede che una mossa votata dalla maggioranza relativa delle istanze possa anche non essere selezionata se il "peso" delle istanze che l'hanno proposta non riflette sufficiente fiducia riguardo la loro scelta: in questo caso i valori dei pesi sono stati fissati in modo assolutamente arbitrario, nel rispetto della sola logica di favorire pesantemente le categorie di conoscenza che occupano le prime posizioni nell'ordinamento precedentemente fissato.

La sperimentazione dei due criteri di selezione ha portato a 32 il numero di giocatori paralleli valutati; complessivamente sono state quindi eseguite 640 partite.

Deve essere precisato che le partite non sono state eseguite in ambiente distribuito, ma simulate attraverso codice scritto completamente in C; i motivi che hanno determinato questa scelta sono duplici:

• evitare, data la durata degli esperimenti (150 minuti ciascuno se eseguiti nel distribuito) ed il loro elevato numero, di creare sovraccarico di lavoro su una grossa parte della rete locale di workstation tale da disturbare altra utenza;

• l'eterogeneità di prestazioni delle workstation avrebbe ingiustamente favorito alcune delle istanze alterando le condizioni ideali di sperimentazione; l'esecuzione simulata degli esperimenti su un unico calcolatore supera ovviamente questo problema.

Gli esperimenti descritti sono stati preceduti da una serie di partite di test volte a valutare la "taratura" di GnuChess, cioè quanto esso favorisca o meno la vittoria del giocatore bianco. Sono state eseguite 110 partite complete in cui GnuChess ha giocato contro se stesso (senza il libro di aperture) ottenendo i seguenti risultati:

• 49% di vittorie del bianco
• 33% di vittorie del nero
• 18% di situazioni di parità.

Tali risultati non si discostano molto dai valori statistici che si riscontrano nella pratica agonistica dei giocatori umani: GnuChess può dunque essere considerato un giocatore "ben tarato" e ciò conferisce maggiore stabilità e validità alle valutazioni sperimentali dei giocatori paralleli precedentemente descritti.

In Tab. 6.4 sono riportati i punteggi ottenuti da ciascun giocatore parallelo; accanto ad essi è distinto il numero di vittorie da esso riportate rispettivamente con il bianco ed il nero.

	Criterio di selezione	
Giocatore parallelo	**Semplice**	**Generalizzato**
3a (conoscenza minima)	10.74 (6+3)	13.30 (7+6)
4a	9.11 (5+3)	13.32 (8+4)
5a	9.91 (6+2)	13.40 (8+5)
6a	9.31 (4+3)	13.53 (6+6)
3b (distribuzione bilanciata)	15.28 (8+7)	11.40 (7+4)
4b	11.45 (3+7)	13.32 (7+4)
5b	11.94 (8+4)	13.26 (6+6)
6b	8.96 (5+3)	14.05 (9+4)
3c (distribuzione sbilanciata)	10.99 (6+4)	13.94 (6+7)
4c	10.46 (4+3)	12.65 (6+5)
5c	10.31 (4+5)	12.48 (5+6)
6c	7.79 (4+3)	9.27 (4+3)

3d (distribuzione incrementale)	11.94 (5+5)	10.21 (6+4)
4d	14.61 (7+6)	11.38 (3+6)
5d	12.43 (7+5)	8.05 (4+2)
6d	7.72 (1+5)	8.12 (5+2)

Tab. 6.4 Valutazione sperimentale dei giocatori paralleli.

I risultati di Tab. 6.4 sono di complessa interpretazione. Alcuni aspetti emergono tuttavia evidenti:

• il 75% dei giocatori paralleli ha ottenuto un punteggio >10 e si è quindi dimostrato più forte di GnuChess; questo risultato è decisamente incoraggiante perché dimostra che in generale la distribuzione della conoscenza determina benefici sulla qualità di gioco;

• è sorprendente il fatto che, tranne in poche eccezioni, i giocatori paralleli risentono negativamente dell'aumento del numero di istanze; il motivo di questo risultato potrebbe essere una distribuzione delle euristiche non oculata per cui l'inserimento di nuove istanze estende l'insieme di mosse proposte favorendo la possibilità di scegliere una mossa non valida.

• nel confronto fra i diversi criteri di distribuzione si è rivelato più efficace quello di distribuzione bilanciata; rispetto agli altri criteri esso prevede che tutte le istanze contengano un sufficiente numero di categorie di conoscenza.

Gli esperimenti di Schaeffer hanno dimostrato che certe categorie sono più efficaci se combinate con altre: il fatto che gli altri criteri stabiliscano istanze con un minore numero medio di categorie di conoscenza può quindi spiegare la minore qualità della loro performance.

• come atteso, il criterio di selezione a maggioranza generalizzato ha conferito maggiore efficacia ai giocatori paralleli; fanno eccezione i giocatori generati con distribuzione incrementale della conoscenza per i quali la versione semplice del criterio a maggioranza si è rivelata migliore. Un possibile motivo di questa anomalia è che i pesi attribuiti alle istanze di tali giocatori con il criterio generalizzato non ne rispecchino la reale forza relativa e quindi favoriscano le scelte delle istanze più deboli.

Le considerazioni appena espresse offrono utili indirizzi per lo sviluppo di ricerche future, ma non costituiscono conclusioni definitive; troppe, infatti,

sono state le scelte arbitrarie operate durante il progetto degli esperimenti: l'ordinamento delle euristiche, la scelta dei criteri di distribuzione e di selezione, la limitazione della durata delle partite e l'attribuzione dei punteggi. A queste deve aggiungersi l'errore statistico indotto dal numero non elevato di partite che hanno caratterizzato ogni torneo.

6.3.2.3 Criteri di selezione a maggioranza generalizzata con pesi variabili

In un criterio di selezione a maggioranza generalizzata, la soluzione più generale per l'attribuzione di pesi è consentirne la variabilità, cioè fissare una loro dipendenza nei confronti di alcune caratteristiche della posizione cui è applicata la scelta.

Intuitivamente l'approccio con pesi variabili si presenta più affidabile perché offre una soluzione ai problemi già denunciati riguardo la variabilità di importanza relativa delle varie euristiche. Quest'ultima soluzione, tuttavia, richiede ulteriore conoscenza del dominio per poter essere applicata. Dalla qualità e dalla complessità di tale conoscenza dipende l'affidabilità del metodo.

Una possibile soluzione è eseguire a priori un'analisi della posizione iniziale (possibilmente con un sistema esperto) per dedurne caratteristiche generali volte a capire quali delle euristiche di valutazione abbiano maggiore importanza in quel caso particolare.

Un metodo molto più semplice ed usato in modo analogo nel calcolo tradizionale della funzione di valutazione, è invece quello di suddividere l'arco della partita in stadi e dedurre lo stadio cui appartiene la posizione corrente dal semplice conteggio dei pezzi sulla scacchiera. In questo caso la determinazione dei pesi risulta comunque approssimativa; inoltre solo per alcune euristiche è possibile stabilire l'importanza relativa che esse hanno in stadi distinti della partita. Nonostante questo il metodo appare più promettente della versione con pesi costanti.

Ciò che è auspicabile è la determinazione di metodi che permettano di stabilire una dipendenza dei pesi delle funzioni di valutazione non solo nei confronti del numero di pezzi, ma anche della loro disposizione sulla scacchiera.

I metodi appena suggeriti sono dipendenti dal dominio di applicazione. In seguito sarà indagata l'efficacia di alcuni criteri con pesi variabili dove l'attribuzione di

questi è indipendente dal dominio e fa riferimento a proprietà generali della ricerca di alberi di gioco: il valore minimax e la profondità di ricerca raggiunta.

6.3.2.3.1 Criteri di selezione basati sui valori minimax delle migliori mosse

Ogni istanza di ricerca produce come risultato la mossa considerata migliore dal suo punto di osservazione ed il valore minimax ad essa associato. In che modo il criterio di selezione può essere condizionato anche da questo valore?

L'applicabilità dei criteri di selezione che verranno descritti in seguito è vincolata dalla veridicità della seguente ipotesi HP_1:

tutte le funzioni di valutazione hanno lo stesso valore di equilibrio (normalmente lo zero) fra i due giocatori e sono tarate in modo che due di esse che ritornano lo stesso valore nella valutazione della stessa posizione intendono esprimere la stessa quantità di vantaggio (o svantaggio) posizionale.

Dato il giocatore parallelo G costituito dalle istanze $I_1, I_2, \ldots, I_n$ si supponga che l'istanza I_1 ritorni la mossa M_1 con valore minimax V_1 ed analogamente per l'istanza I_2 siano la mossa M_2 ed il valore V_2 il frutto della sua ricerca. Si supponga inoltre che la funzione di valutazione di I_1 sia basata sull'euristica E_1 e quella di I_2 sull'euristica E_2.

Sia $V_1 > V_2$. Se è verificata l'ipotesi HP_1 allora la linea di gioco proposta da I_1 conduce ad una posizione in cui il vantaggio rispetto all'euristica E_1 è maggiore del vantaggio rispetto ad E_2 nella posizione cui guida la scelta di I_2. È certamente frettoloso concludere che la scelta di I_1 sia migliore. Il problema è che nel confronto non si è tenuto conto della situazione di partenza, cioè del vantaggio rispetto alle due euristiche nella posizione alla radice della ricerca.

Supponiamo, ad esempio, che E_1 riguardi la struttura pedonale ed E_2 la sicurezza del re. Siano F_1 ed F_2 le rispettive funzioni di valutazione di I_1 ed I_2, P la posizione iniziale e P_1, P_2 le posizioni terminali delle varianti principali delle due istanze.

Siano i seguenti valori: $F_1(P)=10$, $F_2(P)=0$, $V_1=F_1(P_1)=11$, $V_2=F_2(P_2)=8$.

È evidente come in P vi sia una vantaggiosa struttura pedonale, mentre riguardo la sicurezza del re si ha un perfetto equilibrio. Seppure $V_1>V_2$, cioè il vantaggio nella struttura pedonale in P_1 è maggiore di quello nella sicurezza del re in P_2, la variazione relativa alla posizione iniziale $F_i(P_i)-F_i(P)$ calcolata per I_2 è maggiore di quella per I_1. In questa situazione sembra ragionevole preferire la proposta di I_2.

Ipotizziamo di poter valutare a posteriori (conclusa la ricerca delle istanze) i valori $F_1(P_2)$ e $F_2(P_1)$ (è sufficiente che le istanze di ricerca ritornino non solo la miglior mossa, ma l'intera continuazione principale). Siano allora: $F_1(P_2)=4$ e $F_2(P_1)=-2$.

Questi dati svelano il seguente scenario: la mossa M_1 ottiene un miglioramento minimo riguardo la struttura pedonale, ma non degrada (anzi migliora) la sicurezza del re; M_2, invece, conduce ad un grosso miglioramento rispetto alla sua euristica, ma cancella completamente il notevole vantaggio iniziale della struttura pedonale. Queste nuove informazioni capovolgono nuovamente la preferenza nella scelta, a vantaggio della mossa M_1 rivelatasi più "stabile".

L'idea di valutare la variazione relativa alla radice delle valutazioni prodotte dalle istanze di ricerca sembra promettere un valido aiuto alla selezione della migliore mossa, in particolare nella individuazione (e quindi eliminazione) di quelle mosse che seppure migliori secondo certe euristiche celano un comportamento disastroso secondo alcune altre.

6.3.2.3.2 Criteri basati sulla profondità di ricerca delle istanze

In un approccio a pesi variabili il valore dei pesi potrebbe essere variato in funzione della profondità di ricerca raggiunta da ciascuna istanza. In condizioni di torneo è ragionevole affidare ad ogni istanza lo stesso tempo di ricerca. In questo arco di tempo le istanze raggiungono profondità massime dell'albero di gioco in generale diverse. L'idea è di confidare maggiormente nelle ricerche più profonde.

Il problema che può sorgere con questo tipo di soluzione è il seguente: alcune funzioni di valutazione richiedono più tempo per essere calcolate e quindi le istanze relative ad esse saranno quelle che in media raggiungeranno le minori profondità di ricerca; l'effetto potrebbe essere quello di penalizzare (giustamente?) sempre le proposte delle stesse istanze.

6.3.2.4 Criteri di selezione con visita selettiva a posteriori dell'albero di gioco

Un differente approccio per l'implementazione del criterio di selezione è prevedere una ricerca a posteriori dell'albero di gioco le cui mosse al top-level sono le sole proposte dalle istanze; in particolare:

• sia fissata una funzione di valutazione F che tenga conto di un insieme quanto più completo di categorie di conoscenza terminale: potrebbe essere la funzione originale che è stata decomposta oppure una più complessa;

• F è impiegata in una nuova ricerca su albero di gioco dove le mosse alla radice sono limitate a quelle proposte dalle istanze di ricerca (o un loro sottoinsieme già selezionato secondo qualche criterio). La mossa risultata migliore da questa ricerca è quella che sarà effettivamente giocata.

Uno dei parametri di questo criterio è la durata della ricerca a posteriori rispetto a quella delle istanze; nell'assegnamento di tale valore si deve tenere conto del vantaggio di poter disporre, nella nuova ricerca, della cooperazione di tutte le istanze (potrebbe essere impiegato uno degli algoritmi di distribuzione della ricerca presentati nel Capitolo 5).

Una possibile soluzione è vincolare questa nuova esplorazione ad una profondità massima ben inferiore a quella raggiunta dalle singole istanze: in questo caso la funzione F potrà essere anche molto complessa poiché applicata ad un limitato numero di nodi terminali e quindi il maggiore costo del suo calcolo inciderà minimamente sull'efficienza complessiva.

Questo approccio si presenta molto efficace nel riconoscere, fra quelle proposte, mosse scadenti nel complesso; tuttavia appare inaffidabile nel risolvere la scelta fra mosse parimenti valide in quanto questa avverrebbe sulla base della conoscenza di linee di gioco troppo brevi.

Si può quindi pensare di impiegare con efficacia questo metodo in una prima fase di "scrematura" delle mosse candidate. Quando usato per questo scopo il criterio potrebbe essere modificato come segue: per ogni mossa candidata viene calcolato l'esatto valore minimax rispetto ad una profondità prefissata ed alla stessa funzione di valutazione F. In questo caso la valutazione a posteriori di ciascuna mossa è a cura della stessa istanza di ricerca che la ha proposta. In questo modo si ottiene un ordinamento delle mosse in funzione dei rispettivi valori minimax che evidenzierà la grossolana inefficacia di alcune di esse.

La sperimentazione di questo criterio di selezione e di quelli a maggioranza con pesi variabili non avverrà con lo svolgimento di partite complete a causa dell'eccessivo numero che ne deriverebbe. La valutazione dei giocatori avverrà invece attraverso l'esplorazione di un elevato numero di posizioni di test e sarà illustrata in 6.3.3.1; questo metodo, estremamente più rapido, ha comunque notevole valore statistico.

6.3.3 Una valutazione separata dei criteri di distribuzione della conoscenza e dei criteri di selezione

Si consideri un giocatore parallelo G definito per distribuzione della conoscenza; sia M l'insieme di mosse proposte, ad un certo turno di gioco, dalle istanze di G: una situazione sgradita che può presentarsi è che la mossa "migliore" non faccia parte dell'insieme M di candidate.

Il problema è che una mossa è considerata migliore quando massimizza i vantaggi sotto tutti gli aspetti del gioco considerati nel loro complesso e ciò può verificarsi anche se essa non è la migliore in nessuno di questi aspetti considerati singolarmente. Le istanze di G contengono in generale solo una parte della conoscenza terminale e quindi potrebbero facilmente incorrere in questo errore.

La soluzione di questo problema risiede nella possibilità di ogni istanza di ricerca di proporre una graduatoria di mosse alle spalle della migliore. In questo modo anche una mossa ritenuta valida in tutte le ricerche, ma che non risulta la migliore in nessuna di esse, avrebbe qualche possibilità di essere selezionata. Purtroppo le informazioni necessarie per questo tipo di soluzione non possono essere raccolte se si utilizza l'algoritmo $\alpha\beta$, in quanto per ottenere un ordinamento dei valori minimax di tutte le mosse sarebbe necessario

visitare interamente l'albero di gioco (come avviene per il solo algoritmo minimax).

Un possibile criterio di valutazione di un giocatore parallelo potrebbe essere quello di controllare la frequenza con cui la migliore mossa è proposta da almeno una delle sue istanze. Il concetto di migliore mossa è piuttosto astratto ed in generale impossibile rendere oggettivo; una sua buona approssimazione potrebbe essere quella di considerare "migliore" la mossa che giocherebbe il campione del mondo.

Si osservi che la metrica descritta non è influenzata dal criterio di selezione; in realtà essa dipende solamente dalla qualità di gioco delle singolo istanze e non tiene dunque in nessun conto la loro cooperazione a posteriori. Essa esprime dunque le capacità potenziali di un giocatore parallelo: esse potrebbero essere confermate o invece pregiudicate dal criterio di selezione. L'efficacia di un criterio di selezione può essere misurata proprio in base alla sua capacità di fare emergere la mossa migliore nei casi in cui essa è presente nell'insieme di candidate.

Le idee appena proposte saranno applicate in alcuni esperimenti basati sulla esplorazione di posizioni di test.

6.3.3.1 Progetto e risultati di alcuni esperimenti con posizioni di test

Le metriche di valutazione dei criteri di distribuzione e selezione descritte in 6.3.3 sono ora formalizzate per essere applicate in esperimenti basati sulla ricerca di un insieme di posizioni di test.

Sia $P=\{P_1,\ldots,P_n\}$ un insieme di n posizioni di gioco distinte e sia $best(P_i)$ (i=1..n) la mossa "migliore" per la posizione P_i.

Sia G un giocatore parallelo costituito dall'insieme di istanze $I=\{I_1,\ldots,I_k\}$.

È definita la seguente metrica C_d per la valutazione del criterio di distribuzione della conoscenza che ha prodotto G:

• sia $M_G(P_i)=\{m_{i1},\ldots,m_{is}\}$ $(s\leq k)$ l'insieme di mosse scelte dalle istanze di G per la posizione P_i (i=1..n);

• sia d_G: $P\rightarrow\{0,1\}$ la funzione così definita:

$$d_G(p)=\begin{cases}1 & \text{se } best(p)\in M(p)\\ 0 & \text{altrimenti}\end{cases}$$

• infine:

$$C_D(G)=\frac{\sum_{i=1}^{n} d_G(P_i)}{n}$$

Analogamente è definita la metrica C_s di valutazione del criterio di selezione di G:

• sia $Sel_G(P_i)$ la mossa scelta da G dopo l'applicazione del criterio di selezione alle mosse $M_G(P_i)$ proposte dalle sue istanze per la posizione P_i (i=1..n);

• sia la funzione $d_G: P \rightarrow \{0,1\}$:

$$s_G(p) = \begin{cases} 1 & \text{se } Sel_G(p) = best(p) \\ 0 & \text{altrimenti} \end{cases}$$

• la funzione Val calcola il valore complessivo di G normalizzando rispetto al totale delle posizioni il numero di mosse "giuste":

$$Val(G) = \frac{\sum_{i=1}^{n} s_G(P_i)}{n}$$

La definizione della metrica Val è valida anche per giocatori sequenziali e potrà dunque essere usata anche per valutare le prestazioni di GnuChess.

• per stimare l'influenza del criterio di selezione sul valore del giocatore, il valore Val(G) deve essere normalizzato rispetto al "degrado" introdotto dalla distribuzione della conoscenza:

$$C_s(G) = \frac{Val(G)}{C_D(G)} = \frac{\sum_{i=1}^{n} s_G(P_i)}{C_D(G)}$$

In [Ana90] è presentata un'analisi statistica di metriche della stessa natura di quelle appena descritte che ne giustifica la validità (in particolare è indicata una sufficiente correlazione statistica fra queste metriche e la reale qualità di gioco).

Gli esperimenti saranno eseguiti su un insieme di 500 posizioni scacchistiche tratte da [LanSmi93]: sono una raccolta di posizioni relative alla fase di mediogioco appositamente studiate per testare la qualità di un giocatore. Per ciascuna di esse è nota la mossa migliore.

I giocatori paralleli impegnati in questo esperimento sono quelli di Tab. 6.2. Per ridurre il tempo di CPU necessario al completamento dell'esperimento esso non è stato eseguito in ambiente distribuito, ma utilizzando una versione simulata sequenziale dei giocatori paralleli. Una possibile ottimizzazione è la seguente: si osservi che alcune istanze sequenziali sono ripetute in più giocatori paralleli; di conseguenza la ricerca delle posizioni da parte di queste istanze può essere eseguita una sola volta. La prima fase dell'esperimento è dunque quella di isolare le istanze sequenziali che compongono tutti i giocatori paralleli e raccogliere le mosse scelte da ognuno di essi in

seguito alla ricerca di ciascuna posizione; insieme a quest'ultime saranno memorizzati anche tutti i dati statistici (relativi alle ricerche) che saranno necessari all'applicazione, in una fase successiva, dei diversi criteri di selezione.

Il tempo di ricerca di una posizione è stato fissato a 60 secondi.

In Tab. 6.5 è riportata la valutazione delle istanze sequenziali impegnate nella prima fase.

L'esperimento ha confermato che tutte le istanze sequenziali sono singolarmente più deboli di GnuChess; ciò significa che la soppressione di conoscenza terminale dalla funzione di valutazione di GnuChess determina in generale effetti negativi sulla qualità del giocatore sequenziale.

Sarebbe interessante verificare se l'insieme di posizioni "risolte" con successo da ogni istanza sequenziale è un sottoinsieme di quello relativo a GnuChess, cioè se le prestazioni di ogni istanza sono "coperte" da quelle di quest'ultimo.

A questo proposito sono state contate il numero Nt di posizioni in cui almeno uno fra tutti i giocatori di Tab. 6.5 ha individuato la mossa migliore; il valore risultante è sorprendente: Nt=271! Il significato di questo risultato è che in 144 posizioni (≡28.8% del totale) almeno uno dei giocatori con minore conoscenza terminale ha operato una scelta di migliore qualità rispetto a GnuChess. Questo dato statistico è molto importante perché dimostra che l'approccio con distribuzione della conoscenza può potenzialmente raddoppiare le prestazioni del giocatore sequenziale originario; naturalmente il numero delle istanze, la loro identità e il criterio di selezione finale determinano le reali prestazioni del giocatore parallelo.

Istanza sequenziale G	$\sum_{i=1}^{n} S_G(P_i)$	**Val(G)**
(MB)	91	0.182
(MX)	93	0.186
(MK)	78	0.156
(MP)	107	0.214
(MA)	86	0.172
(MR)	83	0.166
(MBX)	107	0.214
(MBXK)	104	0.208

(MBXP)	120	0.240
(MBXA)	111	0.222
(MBKA)	107	0.214
(MBKR)	100	0.200
(MBPR)	101	0.202
(MXKA)	98	0.196
(MXKR)	98	0.196
(MXPA)	123	0.246
(MXAR)	103	0.206
(MKPR)	99	0.198
(MBXKP)	117	0.234
(MBXKPA)	118	0.236
(MBXKPR)	118	0.236
(MBXKPAR) = GnuChess	127	0.254

Tab. 6.5 Valutazione delle istanze sequenziali su 500 posizioni di test.

I dati di Tab. 6.5 forniscono interessanti indicazioni sull'importanza relativa delle categorie di conoscenza terminale di GnuChess:

• l'analisi della struttura pedonale è decisamente l'euristica di valutazione più efficace; contrariamente alle indicazioni di Schaeffer la sua presenza si rivela utile non solo in funzioni di valutazione ricche di euristiche, ma anche se combinata con poche altre;

• la conoscenza con minore importanza relativa appare la sicurezza del re; trova infatti conferma il risultato di Schaeffer secondo il quale questa conoscenza si rivela significativa in pochissime situazioni di gioco. Tuttavia in un ordinamento generale delle categorie di conoscenza essa non può essere inserita all'ultimo posto perché nelle posizioni in cui ha significato questo tipo di analisi la sua presenza può essere decisiva per la vittoria di una partita.

• le altre euristiche di valutazione rispettano sostanzialmente l'ordinamento intuitivo stabilito arbitrariamente in 6.3.2.2.

Nella seconda fase dell'esperimento sono stati valutati i criteri di distribuzione relativi agli stessi giocatori paralleli di Tab. 6.2; in particolare è stato calcolato per ciascuno di essi il valore della metrica C_D. I risultati di questa elaborazione sono riportati in Tab. 6.6.

Etichetta	**Giocatore parallelo G**	$\sum_{i=1}^{n} d_G(P_i)$	$C_D(G)$

3a (minima conoscenza)	(MB)(MX)(MK)	156	0.312
4a	(MB)(MX)(MK)(MP)	196	0.392
5a	(MB)(MX)(MK)(MP)(MA)	207	0.414
6a	(MB)(MX)(MK)(MP)(MA)(MR)	222	0.444
3b (distribuzione bilanc.)	(MBXP)(MBKR)(MXKA)	181	0.362
4b	(MBXA)(MBKR)(MXPA)(MKPR)	207	0.414
5b	(MBXK)(MBKA)(MBPR)(MXKR)(MXPA)	201	0.402
6b	(MBXK)(MBKA)(MBPR)(MXPA)(MXAR)(MKPR)	211	0.422
3c (distribuzione sbilanc.)	(MB)(MX)(MBXKPAR)	182	0.364
4c	(MB)(MX)(MBXKPAR)(MBXKPA)	189	0.378
5c	(MB)(MX)(MK)(MBXKPAR)(MBXKPA)	204	0.408
6c	(MB)(MX)(MK)(MBXKPAR)(MBXKPA)(MBXKPR)	212	0.424
3d (distribuzione incr.)	(MB)(MBX)(MBXK)	127	0.254
4d	(MB)(MBX)(MBXK)(MBXKP)	163	0.326
5d	(MB)(MBX)(MBXK)(MBXKP)(MBXKPA)	171	0.342
6d	(MB)(MBX)(MBXK)(MBXKP)(MBXKPA)(MBXKPAR)	186	0.372

Tab. 6.6 Valutazione separata dei criteri di distribuzione

I dati statistici di Tab. 6.6 indicano che anche giocatori paralleli costituiti da un esiguo numero di istanze possono potenzialmente migliorare in modo sensibile la qualità di gioco di GnuChess.

I criteri di distribuzione a., b. e c. si sono rivelati molto efficaci nella generazione di giocatori paralleli con elevata qualità di gioco (potenziale). È invece confermata la poca validità del criterio incrementale (d.).

Molto interessante è il fatto che il valore maggiore per la metrica C_D sia stato ottenuto dal giocatore (MB)(MX)(MK)(MP)(MA)(MR): ciò sta a significare che la combinazione di giocatori sequenziali con minima conoscenza riesce a catturare la componente strategica più significativa di una posizione di gioco. Rimane purtroppo il problema di come fare emergere il giusto suggerimento fra quelli proposti.

Lo stadio finale dell'esperimento riguarda la valutazione di diversi criteri di selezione applicati agli stessi giocatori paralleli protagonisti della fase precedente; di seguito è riportato un elenco di tali criteri:

• criteri (a maggioranza) "semplice" e "generalizzato": sono gli stessi criteri di selezione descritti nell'esperimento in 6.3.2.2; l'attribuzione dei pesi alle euristiche di valutazione è invariata.

• sia G costituito dalle istanze $I_1,\ldots,I_n$ caratterizzate rispettivamente dalle funzioni di valutazione $f_1,\ldots,f_n$; sia inoltre M una delle mosse candidate e

$I_M=\{I_{M1},\ldots,I_{Mk}\}$ l'insieme di istanze che la ha proposta; saranno sperimentati i seguenti criteri a maggioranza con pesi variabili (differenziati per il criterio di attribuzione del peso w_M):

• valore minimax: sia P la posizione iniziale e P_i la posizione terminale della variante principale calcolata dalla istanza I_i (i=1..n). Sono allora definiti i seguenti criteri di selezione:

$w_M = \sum_{i=1}^{k} [f_{Mi}(P_{Mi}) - f_M(P)]$ (analisi locale del valore minimax)

$w_M = \sum_{i=1}^{k} \sum_{j=1}^{n} [f_j(P_{Mi}) - f_j(P)]$ (analisi globale del valore minimax)

• profondità: il peso di una mossa è funzione della profondità di ricerca raggiunta dalla istanza $I_{Mi} \in I_M$; di norma la ricerca a profondità D_i non è completata (perché interrotta a causa dell'evento "raggiunto tempo max. di ricerca") ed è dunque importante tenere conto del numero N_{mi} di mosse al top-level esplorate completamente.

È inoltre definita una costante C (=100) sufficientemente grande da rendere il parametro Nm decisivo nel confronto fra i pesi di due mosse solo quando la somma delle rispettive profondità massime è la stessa. Si osservi che di fatto il criterio è a maggioranza semplice con soluzione dei casi di parità di voti basata sulle profondità di ricerca:

$$w_M = \sum_{i=1}^{k} (C \cdot D_i + Nm_i)$$

• visita selettiva a posteriori: saranno sperimentate due versioni di questo metodo. Entrambe prevedono una ricerca a profondità massima prefissata (=3-ply). La ricerca è sequenziale poiché, data la limitata profondità, è trascurabile il tempo necessario al suo completamento; la ricerca a posteriori è selettiva perché limitata ad un sottoinsieme delle mosse al top-level. In particolare i due criteri si distinguono per l'identità di tale sottoinsieme:

• il criterio generale prevede che esso coincida con l'insieme di mosse candidate da almeno un'istanza;

• il criterio semplificato prevede invece che la ricerca riguardi solamente le mosse che hanno ottenuto il maggior numero di voti; la ricerca a posteriori è dunque inutile nel caso in cui vi sia un'unica mossa più votata.

	Criteri di selezione													
	Maggioran. semplice		Maggioran. generalizz.		Profondità ricerca		Minimax locale		Minimax globale		Posteriori semplice		Posteriori generale	
G	Val	Cs	Val	Cs	Val	Cs	Val	Cs	Val	Cs	Val	Cs	Val	Cs
3a	0.180	0.577	0.180	0.577	0.184	0.590	0.208	0.667	0.178	0.571	0.182	0.583	0.180	0.577
4a	0.200	0.510	0.200	0.510	0.204	0.520	0.226	0.577	0.174	0.444	0.182	0.464	0.194	0.495
5a	0.210	0.507	0.210	0.507	0.216	0.522	0.228	0.551	0.176	0.425	0.182	0.440	0.200	0.483
6a	0.202	0.455	0.202	0.455	0.212	0.477	0.214	0.482	0.172	0.387	0.182	0.410	0.196	0.441
3b	0.232	0.641	0.232	0.641	0.256	0.707	0.242	0.669	0.220	0.608	0.240	0.663	0.232	0.641
4b	0.244	0.589	0.244	0.589	0.242	0.585	0.242	0.585	0.206	0.498	0.226	0.546	0.224	0.541
5b	0.232	0.577	0.232	0.577	0.232	0.577	0.232	0.577	0.206	0.512	0.210	0.522	0.220	0.547
6b	0.232	0.550	0.232	0.550	0.224	0.531	0.224	0.531	0.210	0.498	0.210	0.498	0.220	0.521
3c	0.200	0.549	0.200	0.549	0.222	0.610	0.248	0.681	0.200	0.549	0.182	0.500	0.200	0.549
4c	0.242	0.640	0.242	0.640	0.242	0.640	0.246	0.651	0.200	0.529	0.182	0.481	0.220	0.582
5c	0.236	0.578	0.236	0.578	0.236	0.578	0.234	0.574	0.174	0.426	0.182	0.446	0.202	0.495
6c	0.238	0.561	0.232	0.547	0.236	0.557	0.232	0.547	0.180	0.425	0.182	0.429	0.202	0.476
3d	0.206	0.811	0.206	0.811	0.214	0.843	0.214	0.843	0.202	0.795	0.186	0.732	0.206	0.811
4d	0.204	0.626	0.204	0.626	0.204	0.626	0.212	0.650	0.198	0.607	0.186	0.571	0.192	0.589
5d	0.216	0.632	0.216	0.632	0.216	0.632	0.216	0.632	0.198	0.579	0.186	0.544	0.210	0.614
6d	0.214	0.575	0.214	0.575	0.216	0.581	0.218	0.586	0.196	0.527	0.186	0.500	0.198	0.532
Valori medi	0.218	0.586	0.218	0.585	0.222	0.598	0.227	0.613	0.193	0.524	0.193	0.521	0.206	0.556

Tab. 6.7 Valutazione dei criteri di selezione e complessiva dei giocatori paralleli.

In Tab. 6.7 sono riportati i dati statistici relativi alla valutazione (metrica Val) dei giocatori paralleli scaturiti dall'applicazione dei criteri di selezione elencati; la metrica C_S, inoltre, consente di stimare l'efficacia di detti criteri indipendentemente dalla distribuzione della conoscenza terminale.

I risultati di Tab. 6.7 sono deludenti: solamente un giocatore parallelo è riuscito a migliorare le prestazioni di GnuChess: il giocatore 3b con criterio di selezione basato sulla profondità di ricerca (lo stesso giocatore che aveva ottenuto la massima valutazione nell'esperimento basato su tornei di partite complete; cfr. Tab. 6.4).

Complessivamente, pertanto, i criteri di selezione proposti non sono riusciti a far emergere (sistematicamente) la mossa migliore, sebbene essa fosse contenuta nell'insieme di candidate.

Dal confronto fra i diversi criteri di selezione appare evidente l'inefficacia dei criteri basati sulla visita

selettiva a posteriori, evidentemente eseguita troppo poco in profondità (solo 3-ply nei nostri esperimenti).

Sorprende la validità dell'approccio basato sul confronto della variazione del valore minimax riferito localmente a ciascuna funzione di valutazione (minimax locale); è però ancora più strana la deludente performance del criterio minimax globale, nonostante esso sia della stessa natura del precedente: è stata quindi negata sperimentalmente l'intuizione descritta in 6.3.2.3.1 secondo cui l'approccio con analisi "globale" delle variazioni dei valori minimax avrebbe dovuto costituire un miglioramento di quello con natura "locale".

6.4 Una rassegna di idee per la distribuzione della conoscenza dirigente

Il significato di distribuire la conoscenza dirigente è combinare in un unico giocatore parallelo più giocatori sequenziali che adottano algoritmi di ricerca diversi.

Gli algoritmi di ricerca su alberi di gioco sono stati classificati in due categorie: di tipo forza bruta e selettivi.

Gli algoritmi della prima classe analizzano tutte le possibili linee di gioco fino ad una profondità massima, mentre quelli selettivi estendono a profondità differenti le varie linee di gioco preferendo quelle più promettenti.

La trattazione seguente intende isolare lo studio della distribuzione della sola conoscenza dirigente: sarà quindi assunto che tutte le istanze contengano la stessa conoscenza terminale e quindi la medesima funzione di valutazione.

6.4.1 Giocatore parallelo con istanze di ricerca di tipo forza bruta

Fra gli algoritmi con forza bruta alcuni sono più efficienti ($\alpha\beta$ contro minimax) e quindi riescono, a parità di tempo, a raggiungere maggiori profondità dell'albero di gioco. Nell'ipotesi di stessa funzione di valutazione, fra due ricerche di tipo forza bruta che raggiungono profondità diverse dell'albero, con molta probabilità sarà più affidabile il risultato della ricerca più profonda.

Se esiste una sola mossa M con il miglior valore minimax (calcolato rispetto alla profondità D) allora tutte le ricerche con forza bruta (qualsiasi sia l'algoritmo) che raggiungono la profondità massima D proporranno la mossa M.

Se invece ci sono più mosse con il miglior valore minimax (rispetto alla profondità D) nulla si può dire su quale, fra queste, sarà scelta dalle istanze con ricerca a profondità D e tantomeno può essere prevista la concordanza delle proposte (queste scelte dipendono dal particolare ordinamento delle mosse nell'albero di gioco).

Sembra allora decisione ragionevole considerare fra gli algoritmi con forza bruta soltanto i più efficienti, vale a dire l'algoritmo $\alpha\beta$ e le sue varianti. Ciò che si aspetta è che non vi sia, fra questi, un algoritmo che vada sempre più in profondità rispetto agli altri, altrimenti il lavoro di quest'ultimi verrebbe sempre scartato.

Questa prima discussione ha già introdotto un necessario criterio di selezione fra le mosse proposte da algoritmi che adottano differente ricerca con forza bruta:

sia $D=\{d_1,d_2,\ldots,d_n\}$ l'insieme di profondità massime raggiunte dalle n istanze $I_1,I_2,\ldots,I_n$ che usano forza bruta e sia $d=\max(d_1,d_2,\ldots,d_n)$. Se $I_{maxdepth}$ è l'insieme delle istanze che hanno raggiunto la profondità d, la scelta finale sarà limitata alle sole mosse proposte dalle istanze in $I_{maxdepth}$.

Supponiamo che gli unici algoritmi usati dalle istanze siano di tipo forza bruta. Intuitivamente il giocatore artificiale risulterà almeno di pari forza rispetto a ciascuna delle sue istanze considerata singolarmente. Scegliendo infatti il criterio di selezione prima descritto il giocatore parallelo giocherà una mossa frutto di una ricerca condotta a profondità maggiore o uguale di quella raggiunta da ognuna delle sue istanze.

Se la congettura appena espressa fosse dimostrata sperimentalmente ciò costituirebbe un risultato di notevole importanza.

È noto come l'efficienza degli algoritmi di tipo $\alpha\beta$ dipenda dalla frequenza dei tagli nella visita e come questa sia strettamente dipendente dall'ordinamento dei nodi. Per alcuni algoritmi questa dipendenza è più marcata, come ad esempio per quelli basati sul concetto di "finestra minimale". Dato che il grado di ordinamento dell'albero di gioco è variabile, in alcune ricerche questi ultimi algoritmi andranno più in profondità rispetto a ricerche $\alpha\beta$ meno influenzate dall'ordinamento dell'albero, mentre in altre (con alberi di gioco non fortemente ordinati) questa situazione sarà ribaltata.

Il giocatore parallelo in analisi tende a superare questa intrinseca dipendenza delle prestazioni dei singoli algoritmi dall'ordinamento dell'albero poiché che esso sarà sempre guidato nella sua scelta dall'istanza scesa più in profondità, quella cioè che ha tratto maggiore vantaggio dal particolare ordinamento dell'albero.

Questo approccio può risultare vantaggioso anche quando le istanze fanno uso diverso della stessa euristica.

Si consideri ad esempio l'euristica "aspiration search" e si supponga che due istanze facciano uso di questa euristica e si differenzino però nella sua applicazione per quanto concerne la dimensione iniziale della finestra $\alpha\beta$. Si ha

quindi un’istanza di ricerca prudente ed una più audace (quella con finestra iniziale più stringente); la fortuna o meno di quest’ultima è legata alla precisione con cui è stato stimato il valore minimax. È noto infatti che se il valore minimax è contenuto nella finestra iniziale, l’istanza che ha iniziato la ricerca con una finestra più stringente ottiene un maggior numero di tagli; se invece esso cade al di fuori di quella finestra l’istanza è costretta a ripetere la ricerca ed in tal caso sarebbe con molta probabilità l’istanza più prudente a rivelarsi più efficiente. La qualità del gioco del giocatore parallelo è comunque indipendente dal grado di precisione della stima del valore minimax in quanto esso è guidato in ogni caso dalla più "fortunata" delle due istanze.

6.4.2 Giocatore parallelo con istanze di ricerca di tipo selettivo

Si consideri ora il caso in cui tutte le istanze di ricerca siano basate su algoritmi di tipo selettivo. In questo contesto appare impredicibile il miglioramento o meno della qualità di gioco del giocatore parallelo rispetto a ciascuna singola istanza.

L’unico criterio di selezione ragionevole è quello a maggioranza generalizzato con pesi costanti. Se fosse nota la maggiore forza di alcune istanze rispetto ad alcune altre, i pesi dovrebbero esprimere una maggiore fiducia verso le mosse proposte da queste istanze di ricerca.

Il metodo con pesi variabili appare invece di difficile applicazione: esso sottintende la conoscenza di una migliore analisi di un’istanza rispetto ad un’altra in funzione della posizione corrente; ciò significherebbe sapere ad esempio che un certo algoritmo selettivo tratta meglio posizioni tattiche piuttosto che strategiche o viceversa. Questa conoscenza è certamente difficile da ottenere.

Un algoritmo selettivo estende soltanto le linee di gioco "più promettenti". La qualità di questi algoritmi dipende dalla probabilità con cui linee di gioco "valide" vengano erroneamente scartate. Questa probabilità dipende principalmente dal criterio di espansione con cui vengono indicate le linee di gioco più promettenti. Ogni algoritmo selettivo ha un differente criterio di espansione e quindi una diversa probabilità di fallimento. Ciò che si aspetta combinando le mosse proposte da più ricerche selettive è di ottenere una riduzione della probabilità di trascurare la linea di gioco "migliore".

Nel seguito verrà definita fallimentare un’istanza che erroneamente considera non promettente la migliore linea di gioco.

Per meglio comprendere l’intuizione illustrata precedentemente si consideri il seguente esempio:

siano I_1, I_2 ed I_3 delle istanze di ricerca con algoritmi di ricerca selettivi; supponiamo che il giocatore parallelo adotti un criterio di selezione a maggioranza semplice. Nella ricerca di una posizione P si immagini che l’istanza I_1 fallisca, mentre ciò non accade per I_2 ed I_3 le quali propongono la migliore mossa; sarà quindi questa ad essere giocata dal giocatore parallelo. In questo caso il fallimento di un’istanza non condiziona il comportamento complessivo.

Il metodo, tuttavia, ha un comportamento impredicibile (con forte probabilità di fallimento) nel caso in cui siano più di una le istanze a fallire: in questo caso il giocatore parallelo tenderà a seguire le scelte delle istanze fallimentari, specialmente se in maggioranza rispetto alle altre; la probabilità che quest’ultima eventualità si verifichi si ritiene comunque che sia molto bassa.

Lo studio dei fenomeni descritti e la verifica delle congetture formulate sembrano di particolare interesse. La verifica sperimentale può essere basata su un confronto fra le mosse proposte dalle singole istanze e dal giocatore parallelo con quella giocata dal campione del mondo. Formulato in questi termini l’esperimento può però condurre a risultati poco significativi e conclusioni frettolose. Infatti sarebbe più corretto distinguere fra situazioni in cui la mossa del campione è stata poco considerata dall’algoritmo selettivo (cioè analizzata poco in profondità) da quelle in cui pur essendo stata considerata "promettente" le è stata preferita un’altra nella scelta finale.

Data la diversità di idee e di struttura alla base degli algoritmi selettivi appare difficile stabilire dei metodi generali per attuare concretamente un’analisi più attenta come quella appena descritta.

6.4.3 Giocatore parallelo misto

Di particolare interesse è l’analisi del comportamento di un giocatore parallelo le cui istanze di ricerca siano sia di tipo forza bruta che selettivo.

Come si può combinare con efficacia le mosse proposte da istanze di ricerca con filosofie così dissimili? Quali relazioni possono esistere fra le mosse da esse proposte? Apparentemente nessuna.

I giocatori artificiali attualmente più forti sono tutti basati su ricerca con forza bruta. Gli algoritmi selettivi, seppur più vicini all’approccio dell’uomo nella scelta della mossa, si rivelano ancora non sufficientemente "robusti" ed affidabili. Apparentemente, quindi, le istanze di ricerca selettiva sembrano offrire un contributo negativo per il giocatore parallelo rispetto a quelle con forza bruta.

Resta aperto il problema di individuare un criterio di selezione non banale come quello a maggioranza semplice o a maggioranza generalizzato con attribuzione "più o meno" casuale dei pesi alle istanze.

Una possibile soluzione può essere la seguente: le istanze di ricerca vengono suddivise in due insiemi a seconda del tipo di ricerca (forza bruta o selettiva); viene inoltre attribuita una priorità diversa a ciascun insieme. La mossa viene scelta sempre fra quelle proposte dalle istanze dell'insieme a maggiore priorità. La funzione delle istanze dell'altro insieme è di risolvere situazioni di eventuale incertezza fra le mosse proposte dall'insieme principale.

La qualità di questo approccio deve essere valutata sia quando l'insieme a maggiore priorità è quello degli algoritmi con forza bruta che quando lo è quello degli algoritmi selettivi.

Quello che si aspetta in questa seconda eventualità è che sia ulteriormente ridotta la probabilità di scelta fallimentare da parte del giocatore parallelo. Supponiamo ad esempio che tutte le istanze con algoritmi selettivi propongano mosse diverse; tenendo conto di queste sole proposte ed utilizzando un criterio di selezione con maggioranza semplice la scelta sarebbe casuale esponendo il giocatore parallelo al rischio di scegliere una mossa scadente, cioè frutto di una ricerca fallimentare.

Con il nuovo criterio presentato, però, la componente casuale è sostituita dai suggerimenti delle ricerche con forza bruta e quindi la probabilità che la scelta finale sia di una mossa poco efficace si attende sia notevolmente ridotta.

Una naturale generalizzazione di questo criterio prevede che la suddivisione delle istanze nei due insiemi segua un criterio qualsiasi. Ad esempio, conoscendo la forza relativa delle singole istanze si potrebbero inserire nell'insieme a maggiore priorità le istanze più forti.

Le due partizioni dell'insieme delle istanze appena descritte appaiono le uniche con un fondamento logico. La qualità di queste e di qualsiasi altro tipo di partizione deve essere comunque verificata da esperimenti reali.

Nella presentazione di questi metodi sono stati già discussi diversi criteri di selezione. Nessuno di questi è basato sul valore minimax ritornato dalle istanze. Non si ritiene possa essere stabilita alcuna relazione significativa fra i valori minimax calcolati da istanze che usano algoritmi di ricerca diversi (seppure esse abbiano in comune la stessa funzione di valutazione). Il fatto che il valore dedotto da un'istanza sia maggiore di quello ottenuto da un'altra non permette certo di concludere che la mossa da esse proposta sia migliore.

6.5 Conclusioni

In questo capitolo abbiamo discusso vari aspetti di un nuovo approccio al progetto di giocatori artificiali basato sul concetto di "distribuzione della conoscenza".

Sulla base della classificazione della conoscenza in terminale e dirigente [Ber82] abbiamo decomposto in due parti il problema della sua distribuzione, cioè della cooperazione di più giocatori artificiali caratterizzati, appunto, da diversa conoscenza del dominio di applicazione.

Abbiamo individuato alcune caratteristiche generali del problema ed in particolare due tipi di criterio che sono alla base della generazione dei giocatori paralleli in analisi:

• il criterio di distribuzione, cioè l'insieme di regole che guidano, a partire da una conoscenza di base del dominio, l'attribuzione di porzioni di essa ai giocatori sequenziali che si intende far cooperare;

• il criterio di selezione, cioè il principio che governa la scelta di una fra le mosse proposte da ciascun giocatore sequenziale.

In particolare abbiamo approfondito il problema della distribuzione della conoscenza terminale. A riguardo si è cercato di sviluppare un insieme di metodi e criteri indipendenti dal dominio di applicazione.

La sperimentazione (nel dominio degli scacchi) delle idee emerse nel corso della discussione ha evidenziato la validità dell'approccio: il giocatore sequenziale GnuChess è stato sconfitto da molti giocatori paralleli ottenuti dalla distribuzione della sua stessa conoscenza terminale.

L'evidenza sperimentale ha dimostrato una curiosa anomalia dei metodi di distribuzione della conoscenza terminale: non sempre la disponibilità di un maggior numero di risorse di elaborazione è sinonimo di migliori prestazioni. La motivazione di tale anomalia è in parte dovuta ad alcune proprietà della conoscenza terminale, già individuate in [Sch86], e confermate dai nostri esperimenti:

• esiste un ordinamento per importanza relativa delle categorie di conoscenza terminale;

• esistono categorie di conoscenza terminale la cui efficacia dipende dalla presenza o meno di altre categorie.

Chiaramente la disponibilità di un maggior numero di giocatori sequenziali può favorire la presenza di combinazioni non efficaci delle categorie di conoscenza, determinando così un apporto negativo (e determinante) da parte di alcuni di essi.

La sperimentazione dei giocatori con conoscenza distribuita su un insieme di posizioni di test ha permesso di indagare in profondità le proprietà dei criteri di distribuzione e di selezione.

In generale è emersa l'esigenza di scelte guidate (almeno in parte) dal dominio di applicazione: data l'inerente dipendenza del concetto di conoscenza dal dominio applicativo, un approccio alla sua distribuzione completamente indipendente dal dominio difficilmente può garantire prestazioni ottimali.

L’esigenza appena denunciata si rivela particolarmente necessaria nella definizione dei criteri di selezione. Il principale problema dei criteri suggeriti in questo lavoro è la loro eccessiva semplicità ed incapacità di individuare la reale importanza relativa che deve essere attribuita alle proposte dei singoli giocatori sequenziali. In particolare è apparsa evidente la necessità di criteri di selezione dinamici, cioè capaci di adeguarsi alle caratteristiche dello stato corrente del gioco: lo sviluppo di criteri di questa natura richiede forzatamente che essi incorporino una qualche conoscenza del dominio.

I risultati sperimentali hanno tuttavia sottolineato l’enorme potenzialità di questo nuovo approccio: nel 28.8% delle 500 posizioni di test almeno una delle istanze del giocatore sequenziale originario (GnuChess), pur disponendo di minore conoscenza terminale, ha operato una scelta di migliore qualità rispetto a quest’ultimo. Rimane tuttavia aperto il problema, legato al criterio di selezione, di come far emergere tali migliori suggerimenti.

Capitolo 7
Conclusioni e lavori futuri

I sistemi di reti di calcolatori autonomi, ognuno con la propria memoria privata, sono ormai uno strumento diffuso di elaborazione in virtù del loro eccellente rapporto prezzo/prestazioni; tuttavia, il problema della programmazione di singole applicazioni basate su questi sistemi è ancora aperto [Bal90].

Il principale contributo offerto dal presente lavoro è stato quello di avere impegnato un nuovo modello di programmazione distribuita (Linda) in un campo reale di applicazione (il dominio dei giochi) nel tentativo di evidenziare le principali proprietà di tale modello.

Il dominio dei giochi (ed in particolare degli scacchi) si è rivelato uno strumento di analisi completo:

• la realizzazione di alcuni degli algoritmi classici di visita parallela di alberi di gioco ha permesso di risolvere problemi con differente granularità del parallelismo (Capitolo 5);

• la parallelizzazione di un giocatore sequenziale reale (GnuChess) ha favorito un esame delle capacità del linguaggio di coordinamento Linda di sostenere il riuso di moduli sequenziali nello sviluppo di programmi paralleli (Capitolo 6).

Obbiettivo non meno importante di questo lavoro era quello di fornire un apporto alla ricerca nel campo dei sistemi "intelligenti" attraverso:

• la sperimentazione in ambiente distribuito di algoritmi paralleli di ricerca su alberi di gioco;

• il progetto e la realizzazione di un giocatore basato sul nuovo concetto di distribuzione della conoscenza.

Di seguito sarà valutato il raggiungimento degli obbiettivi prefissi, tracciate alcune conclusioni ed infine proiettato lo sguardo verso gli sviluppi futuri della ricerca.

7.1 Valutazione del raggiungimento degli obbiettivi e conclusioni

7.1.1 Linda

Espressività ed efficienza sono alcune delle principali proprietà di un linguaggio di programmazione distribuita: la valutazione di Linda che emerge dal presente lavoro è in riferimento proprio a tali caratteristiche.

Per espressività di un linguaggio distribuito si intende la facilità offerta al programmatore di esprimere il parallelismo, la comunicazione e la sincronizzazione.

Il parallelismo in Linda è basato sulla creazione esplicita dei processi. Ciò consente estrema flessibilità nella costruzione dell’architettura logica di comunicazione. Nel Capitolo 5 sono state sperimentati due tipi di architettura (relativamente simili) basati rispettivamente sui concetti di cooperazione master-worker (cfr. 5.2.1.1.1) e alla pari (cfr. 5.2.1.2.2): un

processo master distribuisce lavoro che è eseguito da un insieme di processi worker indistinguibili. La differenza fra i due approcci sta nel fatto che nel primo caso la distribuzione del lavoro è dinamica, mentre nel modello alla pari il lavoro che deve essere distribuito è determinato prima della creazione dei worker.

Il modello di comunicazione in Linda, basato sull'implementazione di strutture dati distribuite su uno spazio di tuple, ha permesso in modo molto naturale la realizzazione di tali forme di cooperazione in cui i processi interagiscono anonimamente attraverso un'area logica comune.

Non sempre la condivisione di strutture distribuite è necessaria: per alcune applicazioni il modello a scambio di messaggi, ad esempio, sarebbe la soluzione migliore; tuttavia, quando richiesto, lo scambio di messaggi in Linda può essere simulato attraverso il passaggio di tuple (ciò è in parte avvenuto nel progetto del giocatore con conoscenza distribuita: cfr. 6.3.1).

Le condizioni di sincronizzazione sono espresse attraverso gli stessi operatori di lettura in e rd i quali hanno la proprietà di essere bloccanti, cioè di sospendere il processo fino al verificarsi di un evento (manifestato dalla inserzione nello spazio delle tuple della tupla attesa). Grazie a questa proprietà del linguaggio la programmazione della maggior parte delle sincronizzazioni è molto facile perché implicita nella descrizione dell'azione che consegue al verificarsi dell'evento: ad esempio un processo worker si sincronizza con il master semplicemente "tentando" di prelevare un nuovo lavoro. Vi sono tuttavia situazioni frequenti in cui la sincronizzazione deve essere programmata esplicitamente, ad esempio nella definizione delle barriere di sincronizzazione (cfr. 3.4.2.2); la descrizione di queste situazioni (cfr. 5.2.1.1.2; 5.2.1.4.2; 5.2.2.1; 6.3.1) è risultata nel corso dello sviluppo dei programmi spesso origine di errori di programmazione, sintomo questo che le primitive Linda forse costituiscono strumenti troppo a "basso livello" per questi scopi.

Una proprietà espressiva molto rilevante di Linda è che l'indivisibilità delle sue primitive ha reso trasparente la sincronizzazione dei processi per l'accesso in mutua esclusione alle strutture dati distribuite. Una lacuna del linguaggio è tuttavia l'assenza di meccanismi che garantiscano implicitamente l'indivisibilità nell'accesso ad una molteplicità di strutture dati distribuite. Questa situazione si è verificata, ad esempio, nella implementazione dell'algoritmo base con condivisione dello score (cfr. 5.2.1.4.1): l'aggiornamento dello score e della mossa

migliore devono avvenire in un’unica sezione critica. Anche la descrizione di queste situazioni può originare errori di programmazione (generalmente risultanti a tempo di esecuzione nello stallo dei processi); la soluzione generale di questo problema è delimitare la sezione critica con tuple-semaforo (cfr. 3.4.1.1).

Il progetto del giocatore parallelo basato sulla distribuzione della conoscenza (cfr. 6.3.1) ha messo in evidenza la notevole espressività di Linda quando impegnato nel coordinamento di moduli sequenziali predefiniti; il numero di linee di codice C-Linda, estremamente esiguo rispetto a quello del programma sequenziale originario (GnuChess), sottolinea questa proprietà. In questo senso Linda si rivela uno strumento di programmazione di straordinaria efficacia. Il tipo di approccio alla programmazione parallela che scaturisce da queste considerazioni è decisamente vantaggioso: il programmatore riduce la sua attività alla sola descrizione del coordinamento fra i processi ed è esonerato dalla definizione della loro parte sequenziale.

Il riuso del software sequenziale è risultato difficoltoso nella realizzazione degli algoritmi paralleli discussi nel Capitolo 5. Il motivo di tale complicazione non deve essere comunque attribuito alla espressività del linguaggio di coordinamento, ma ad una non completa modularità del programma sequenziale di partenza (GnuChess). Questo programma, infatti, non è una libreria di funzioni (come in realtà è stato utilizzato in questa fase del lavoro): le sue scelte di progetto sono mirate alla massimizzazione delle prestazioni in sacrificio, talvolta, della modularità del programma.

Si deve quindi concludere che un requisito fondamentale affinché risulti vantaggioso il riuso dei programmi sequenziali è la loro modularità. In generale questo requisito è tanto più necessario quanto più fine è la granularità dell’applicazione parallela che si intende realizzare: la non completa modularità di GnuChess non ha infatti "disturbato" il progetto del giocatore con conoscenza distribuita (caratterizzato da granularità del parallelismo estremamente grossa), ma applicazioni a grana più fine quali sono gli algoritmi di decomposizione discussi nel Capitolo 5.

Le critiche più pesanti formulate in letteratura nei confronti di Linda riguardano l’efficienza: l’indirizzamento associativo delle tuple e la visibilità globale dello spazio omonimo hanno indotto molti studiosi a ritenere che Linda non possa essere implementato efficientemente [Bal92]. Questo lavoro ha in parte confutato questa convinzione: la sperimentazione di alcuni degli algoritmi di decomposizione ha prodotto notevoli speedup rivelando che una programmazione accorta, che tenga conto delle proprietà del linguaggio, può mantenere le

prestazioni delle applicazioni parallele al di sopra di limiti già soddisfacenti (cfr. 5.3).

Questo lavoro ha evidenziato la notevole influenza che ha la particolare implementazione dello spazio delle tuple sulle performance dei programmi:

• l'implementazione con "out distribuita" favorisce le prestazioni delle applicazioni con infrequenti aggiornamenti delle strutture dati distribuite rispetto al numero delle loro letture;

• l'implementazione con "in distribuita", invece, riduce l'overhead determinato dalla generazione di nuove tuple, ma rende lineare nel numero di processi il costo della loro lettura.

L'implementazione di Network C-Linda, l'istanza del modello Linda utilizzata in questo lavoro, è basata sullo schema con "in distribuita". Ciò non ha certo favorito l'efficienza degli algoritmi di decomposizione i quali, per loro natura, prevedono un occasionale aggiornamento delle strutture condivise (finestra $\alpha\beta$) contro una loro frequente lettura; tuttavia in 5.2.1.4.1 è stata presentata una tecnica di programmazione (uso di tuple "private") che ha permesso di limitare l'overhead determinato dagli accessi in lettura allo spazio delle tuple.

La programmazione in Linda di applicazioni che rispettino il requisito di efficienza non può quindi prescindere dalla conoscenza dell'implementazione dei costrutti del linguaggio. Questa realtà limita la libertà del programmatore il quale è costretto ad utilizzare uno stile di programmazione non trasparente rispetto ai livelli inferiori dell'architettura del sistema.

Questo lavoro ha dimostrato che Linda è uno strumento efficace per la realizzazione efficiente di applicazioni con grossa e media granularità del parallelismo. L'implementazione dell'algoritmo generale di decomposizione dinamica (cfr. 5.2.2.2), caratterizzato da una grana più fine del parallelismo, ha rivelato che Network C-Linda non è adatto alla programmazione di questa classe di applicazioni a causa dell'eccessivo overhead delle comunicazioni.

Alcune considerazioni sono infine necessarie riguardo l'overhead determinato dalla creazione dei processi: questa attività si è rivelata molto costosa nel sistema usato in questo lavoro. In 5.2.1.1.2 è stata discussa la conseguente esigenza di limitare il numero di tali operazioni attraverso la definizione di una struttura di processi con funzionalità generali, capaci di specializzarsi qualora venga loro richiesta una modifica dinamica delle proprie funzioni.

7.1.2 La distribuzione della ricerca

Nel corso di questo lavoro sono stati presentati molti algoritmi di ricerca distribuita su alberi di gioco, con particolare attenzione ai metodi di decomposizione dell'albero.

A riguardo è stata formalizzata una nuova classificazione di tali metodi e introdotta una nuova terminologia. Attraverso questi strumenti si è raggiunto l'obbiettivo di stabilire un maggiore ordine all'interno di questa così complessa classe di algoritmi e di fissare dei punti di riferimento per l'individuazione delle loro qualità salienti.

L'evidenza sperimentale ha confermato il valore relativo che è attribuito in letteratura a tali algoritmi: PVSplit ed in particolare la sua variante DPVSplit costituiscono i metodi più efficienti di ricerca parallela $\alpha\beta$ in presenza di un numero di processori limitato (≤ 11) e di un'architettura parallela di base di tipo rete locale. È doveroso sottolineare come Linda abbia aiutato nello sviluppo degli algoritmi, permettendo di applicare selettivamente o incrementalmente nuovi miglioramenti all'algoritmo (estremamente semplice) di partenza (cfr. 5.2.1.1). Fra questi quello più sorprendente in termini di prestazioni è risultato il metodo di condivisione dello score (cioè del limite α al top-level dell'albero): tale condivisione attuata attraverso la memorizzazione dello score in una struttura dati distribuita ha garantito una notevole riduzione dell'overhead di ricerca con un costo delle comunicazioni trascurabile rispetto al guadagno di efficienza complessivo.

L'approccio con decomposizione dinamica si è rivelato efficace solo in combinazione con PVSplit (da cui origina l'algoritmo DPVSplit). La sua finalità, infatti, è di favorire il bilanciamento del carico di lavoro fra i processi aumentando la decomposizione dell'albero: ciò determina però un aumento del numero di comunicazioni inter-processo. Mentre in DPVSplit il vantaggio garantito dalla maggiore uniformità nell'attribuzione del carico è superiore all'overhead di comunicazione addizionale, quest'ultimo diviene insopportabile in algoritmi con maggiore decomposizione come quelli discussi ad epilogo del Capitolo 5.

Gli esperimenti hanno inoltre dimostrato che, qualora sia applicabile, il meccanismo di cooperazione alla pari è preferibile al master-worker (ciò è almeno vero per la programmazione in Linda). Il primo approccio, infatti, ha il pregio di impegnare tutte le risorse di elaborazione in attività produttiva (nella fattispecie nella ricerca di sottoalberi di gioco) distribuendo fra esse i compiti "gestionali". È stato tuttavia spiegato che tale asserzione è vera in presenza di un'architettura con pochi processori: la differenza (in termini

di efficienza) fra i due approcci tende ad annullarsi all'aumentare dei processori.

È stato inoltre presentato un nuovo approccio allo sviluppo di algoritmi $\alpha\beta$ di decomposizione strutturato in due fasi consecutive:

• selezione del criterio di decomposizione più "promettente"
• sviluppo di un algoritmo che tenga conto delle caratteristiche del criterio di decomposizione individuato nella prima fase così da massimizzare le prestazioni.

È stato progettato e realizzato uno strumento di assistenza alla prima delle due fasi elencate: un algoritmo generale di decomposizione dinamica parametrico rispetto al criterio di decomposizione. Tale struttura software deve essere quindi intesa come uno strumento di ricerca che consente di stimare con immediatezza le prestazioni di nuovi criteri di decomposizione, senza che sia programmata ogni volta la struttura di interazioni fra i processi.

7.1.3 La distribuzione della conoscenza

In questo lavoro è stato presentato un approccio relativamente nuovo al progetto di giocatori artificiali paralleli denominato "distribuzione della conoscenza". L'idea alla base dei giocatori così definiti è la cooperazione fra giocatori sequenziali con diversa conoscenza del dominio di applicazione (la cui attribuzione è in rispetto di un criterio detto di distribuzione): ogni giocatore propone la rispettiva migliore mossa e, dall'insieme così ottenuto di candidate, viene scelta (attraverso un criterio detto di selezione) quella che sarà effettivamente giocata.

In riferimento alla classificazione della conoscenza (dovuta a Berliner [Ber82]) in terminale e dirigente sono stati separati i concetti di distribuzione dei due differenti tipi di conoscenza.

Il problema della distribuzione della conoscenza terminale è stato indagato in profondità, mentre relativamente alla distribuzione della conoscenza dirigente sono state suggerite alcune interessanti soluzioni intuitive.

Alcuni esperimenti hanno confermato precedenti risultati [Sch86]:

• esiste un ordinamento per importanza relativa delle categorie di conoscenza terminale;

• esistono categorie di conoscenza terminale la cui efficacia è dipendente dalla presenza o meno di altre categorie.

Nel corso del lavoro è stata generata, su base intuitiva, una rassegna di criteri di distribuzione e di selezione.

Sono stati così definiti dei giocatori paralleli reali le cui prestazioni sono state confrontate in condizioni di torneo contro un avversario comune: il giocatore sequenziale di scacchi GnuChess. I risultati ottenuti sono sorprendenti: il 75% dei giocatori paralleli è risultato vincente; l'obbiettivo di

migliorare le prestazioni di un giocatore sequenziale utilizzando una forma di distribuzione della sua stessa conoscenza è stato quindi eccellentemente raggiunto.

Sono state inoltre definite delle metriche per la valutazione separata dei criteri di distribuzione e di selezione. Questa valutazione è stata applicata sulla base delle prestazioni ottenute dai diversi giocatori paralleli nella esplorazione di un notevole numero (=500) di posizioni scacchistiche di test. Le conclusioni cui conducono tali esperimenti sono chiare:

• i giocatori paralleli con conoscenza distribuita sono "potenzialmente" in grado di raddoppiare le prestazioni (su posizioni di test) del giocatore sequenziale originario. Questa asserzione è basata sulla misura della capacità, di almeno una delle istanze che compongono i giocatori paralleli, di suggerire la mossa realmente migliore in una data posizione. Si parla di prestazioni potenziali proprio perché il suggerimento corretto deve ancora emergere, attraverso l'applicazione del criterio di selezione, come la scelta finale del giocatore parallelo.

• il criterio di selezione è la componente più delicata dei giocatori con conoscenza distribuita. I criteri proposti hanno palesato prestazioni complessivamente deludenti che hanno denunciato la loro eccessiva semplicità ed incapacità di modificare dinamicamente l'importanza relativa attribuita alle proposte delle istanze sequenziali.

7.2 Lavori futuri

7.2.1 Linda

Sebbene vasta e circonstanziata la valutazione del modello Linda di programmazione distribuita operata in questo lavoro ha fissato tre importanti parametri: l'architettura parallela (rete locale di workstation), l'implementazione dello spazio delle tuple ("in distribuita") ed il dominio di applicazione (i giocatori artificiali di scacchi). Sarebbe di sicuro interesse per la comunità scientifica estendere ad ampio spettro (rispetto ai parametri appena citati) l'analisi di questo nuovo modello.

Particolare attenzione dovrebbe essere rivolta all'esame delle sue prestazioni in architetture a parallelismo massiccio. Implementazioni di Linda su ipercubo [BjCaGe89] e transputer [Zen90] sono già state realizzate; tuttavia le applicazioni parallele sviluppate su tali sistemi sono ancora in numero troppo esiguo per poter trarre conclusioni sulla validità del modello Linda quando basato su queste architetture.

Nel corso di questo lavoro è stata più volte sottolineata la dipendenza delle prestazioni delle applicazioni in Linda dalla implementazione delle sue primitive. Questa realtà è

conseguenza della scelta del modello Linda di rendere trasparente al programmatore la distribuzione fisica delle tuple.

La sperimentazione di alcuni algoritmi paralleli di ricerca $\alpha\beta$ ha dimostrato che l'utilizzo di particolari tecniche di programmazione (nel nostro caso "tuple private") può ridurre il degrado delle prestazioni indotto da una implementazione del linguaggio non ottimale per il tipo di applicazione che si intende realizzare. Sarebbe desiderabile uno studio più approfondito di tali problemi che permetta di generare tecniche di programmazione Linda (mirate all'efficienza dei programmi) tali da coprire la casistica di tutte le possibili implementazioni del modello. Ciò favorirebbe la portabilità dei programmi Linda: il programmatore sarebbe già in grado di prevedere quali modifiche apportare alla sua applicazione (volendo impedire un degrado delle prestazioni) quando essa sarà trasportata su un sistema che attua una differente implementazione del modello.

È tuttavia lecito porre alcuni interrogativi sulla possibilità di arricchimento del modello Linda di strumenti che risolvino a più basso livello i problemi descritti:

• quale implicazioni (sull'implementazione del modello e sull'efficienza dei programmi) avrebbe offrire al programmatore la possibilità di disporre del controllo dell'allocazione fisica delle tuple? Un recente approccio in tale direzione è quello offerto dal sistema LIPS per reti di workstation [RotSet93]: si tratta di un sistema ispirato al modello di comunicazione Linda in cui l'invocazione della primitiva out prevede che uno dei parametri specifichi il processore su cui la tupla dovrà essere memorizzata.

• è ragionevole pensare ad un sistema Linda che adatti dinamicamente l'implementazione delle sue primitive in funzione di statistiche raccolte a tempo di esecuzione?

Si potrebbe pensare, ad esempio, di arricchire il nucleo di un sistema Linda implementato con "in distribuita" di funzionalità che consentano comunque la "diffusione" di quelle strutture dati distribuite (tuple) accedute molto frequentemente in lettura ed aggiornate di rado.

L'implementazione del modello master-worker ha evidenziato l'esigenza, per questo tipo di cooperazione, di un controllo a livello di programma dell'allocazione dei processi. Allo stato dell'arte non ci risulta la presenza di sistemi Linda reali che garantiscano questa funzionalità.

7.2.2 Giocatori artificiali paralleli

Questo lavoro ha affrontato molti dei temi legati alla definizione di sistemi "intelligenti" distribuiti, cioè strutture di calcolatori autonomi che cooperano nel prendere decisioni. Le componenti logiche di un sistema intelligente sono la

conoscenza del dominio di applicazione e le capacità di ricerca: il problema della distribuzione di entrambe queste proprietà è stato studiato separatamente lasciando aperta la discussione riguardo una loro combinazione.

Data la complessità dell'argomento sono state esplorate in profondità solo una parte delle problematiche ad esso inerenti ed in taluni casi esse non sono state esaminate in modo esauriente.

In riferimento alla distribuzione della ricerca la discussione ha concentrato la sua attenzione sugli algoritmi di decomposizione, cioè la classe di metodi di ricerca distribuita che ha garantito nel recente passato le migliori prestazioni; è quindi doveroso rivolgere brevemente lo sguardo verso il futuro delle altre forme di parallelizzazione della ricerca su alberi di gioco.

Fra gli approcci alternativi ai metodi di decomposizione quello basato sulla ricerca con finestre parallele è quello maggiormente studiato in passato e non sono attualmente prevedibili ulteriori approfondimenti in questa direzione.

Deep-Thought [AnCaNo90] e HITECH [Ebe87] sono esempi di giocatori paralleli che utilizzano hardware ad hoc per l'implementazione di funzionalità (della ricerca) applicate localmente ad un nodo dell'albero: il calcolo della funzione di valutazione e la generazione delle mosse. Naturalmente il progetto di tale hardware deve essere guidato dal dominio di applicazione. È ragionevole pensare alla definizione in un immediato futuro di metodi generali di progetto di tali componenti che possano essere applicati in più domini?

L'approccio alternativo forse più affascinante e che sembra promettere le migliori prestazioni è quello che stabilisce il mapping degli stati della ricerca nell'insieme dei processori (cfr. 4.2.4); il suo successo è legato agli sviluppi futuri della tecnologia: l'overhead di comunicazione che esso in generale determina è, allo stato dell'arte, insostenibile [FeKoPo93]. Tuttavia un'auspicabile riduzione dei costi delle comunicazioni fisiche inter-processore potrebbe fare di questa classe di metodi la migliore soluzione per molti domini applicativi.

Per quanto concerne gli algoritmi di decomposizione sperimentati in questo lavoro molti di essi non stati valutati completamente; in particolare:

• il metodo di condivisione dello score è stato applicato unicamente al top-level (dove tuttavia si attende che esso manifesti maggiormente la sua efficacia). Quali vantaggi avrebbe apportato la condivisione (in questo caso di entrambi i limiti α e β) anche ai livelli inferiori dell'albero? Fino a quale profondità i vantaggi della condivisione non sarebbero stati vanificati dall'aumento delle comunicazioni?

L’estensione in profondità di tale tecnica avrà incidenza differente su PVSplit rispetto a quella sull’algoritmo base?

• stesse considerazioni devono essere fatte per la versione dell’algoritmo base con subappalto dinamico della ricerca (cfr. 5.2.2.1), applicato anch’esso unicamente al livello superiore dell’albero di gioco.

• fra le euristiche di ordinamento interno dei nodi (cfr. 1.3.3) sono state sperimentate unicamente la tecnica di approfondimento iterativo in combinazione con il riordinamento statico (fra due iterazioni successive) dei nodi al top-level (cfr. 5.2.1.3). Come noto l’incidenza di queste euristiche sull’efficienza della ricerca è notevole [Sch86]. In 4.3 è stata anticipata una discussione generale sugli effetti dell’applicazione di tali tecniche in ambiente distribuito; sarebbe tuttavia interessante verificare tali intuizioni in un ambiente di programmazione Linda. La totalità di queste euristiche è basata sulla gestione di tabelle: la programmazione di quest’ultime come strutture dati distribuite e della loro manipolazione attraverso le primitive Linda sarebbe immediata. Restano però perplessità sull’efficienza della loro gestione, specialmente per la tabella history (8196 posizioni in GnuChess) e quella delle trasposizioni (2^{16} posizioni in GnuChess), le quali, oltre che di elevate dimensioni, sono accedute in corrispondenza di ogni nodo interno dell’albero di gioco e quindi molto frequentemente. Solo la sperimentazione di tali euristiche può indicare con chiarezza se la condivisione di tali tabelle è vantaggiosa ed eventualmente per quali loro parametri (dimensione e numero di accessi). Deve essere comunque valutata l’alternativa di una implementazione di tabelle locali ai processi oppure di un approccio misto che preveda la coesistenza di tabelle globali e locali (quest’ultime aggiornate periodicamente).

Durante lo studio dei metodi di decomposizione dell’albero di gioco è sorta l’esigenza di un loro riordinamento logico: da essa è nata la classificazione presentata in 4.2.5.1. Nel settore della ricerca interessato allo studio di algoritmi paralleli di visita di alberi di gioco deve purtroppo essere segnalata una certa povertà di strumenti formali: molte delle nozioni sono di natura empirica o intuitiva ed è indubbia la difficoltà di tramutare queste in definizioni formali o in oggetto di studi analitici. Il formalismo introdotto in questo lavoro è soltanto un timido approccio in questa direzione: lo sviluppo di strumenti formali ben più sofisticati potrebbe essere la chiave di un progresso sistematico in questo settore.

L’algoritmo generale di decomposizione dinamica proposto alla fine del capitolo 5 rappresenta un esempio di sviluppo di

uno strumento di assistenza per i ricercatori capace di garantire una rapida stima delle prestazioni di algoritmi basati su diversi criteri di decomposizione. La versione corrente di tale algoritmo prevede che la specializzazione del criterio di decomposizione sia attuata modificando il codice di una funzione C-Linda; un lavoro futuro è quello di rendere trasparente l'aggiornamento di tale codice, ad esempio definendo un meta-linguaggio che permetta di descrivere il criterio di decomposizione relativamente ad un certo numero di parametri della ricerca.

Gli algoritmi di decomposizione dinamica si sono rivelati complessivamente inefficienti a causa della granularità più fine del parallelismo la quale ha originato un insostenibile overhead di comunicazione. Sarebbe interessante verificare l'efficienza di questi algoritmi in architetture a parallelismo massiccio dove il costo delle comunicazioni incide in minor misura sulle prestazioni dei programmi.

L'assoluta novità dei metodi di decomposizione della conoscenza introdotti in questo lavoro schiude le porte di un settore finora inesplorato della ricerca e che i risultati sperimentali documentati in questa tesi rendono meritevole di estrema attenzione.

Sono stati fornite molte indicazioni per quanto riguarda la distribuzione della conoscenza terminale; tuttavia molte di esse sono frutto dell'applicazione di idee intuitive che meriterebbero un'indagine più sistematica.

Per quanto concerne i criteri di distribuzione della conoscenza terminale, infatti, sono stati sperimentati solo 4 di essi: è desiderabile fissare nuovi criteri che definiscano altre combinazioni significative della conoscenza all'interno delle istanze del giocatore parallelo. In particolare si dovrebbe tenere in maggiore considerazione gli effetti della combinazione di certe categorie di conoscenza e non lasciare al caso la loro contemporanea presenza all'interno di una stessa istanza.

Indicazioni più complete potrebbero essere ottenute verificando le prestazioni di giocatori risultanti dalla distribuzione della conoscenza ricavata:

- dalla funzione di valutazione di un giocatore reale diverso da GnuChess
- dalla combinazione delle funzioni di valutazione di più giocatori reali
- dalla definizione di una funzione di valutazione ad hoc (finalizzata alla sua "distribuzione").

Già in sede di conclusioni è stato sottolineato il delicato ruolo del criterio di selezione; sono stati sperimentati diversi criteri indipendenti dal dominio di applicazione, che però gli esperimenti con posizioni di test hanno rivelato

insoddisfacenti. Lavori futuri dovranno quindi impegnare anche criteri di selezione più complessi e maggiormente legati al dominio.

L’efficacia di un criterio di selezione è legata alla sua capacità di riconoscere, posizione per posizione, l’importanza relativa dei suggerimenti proposti dalle diverse istanze di ricerca che compongono il giocatore parallelo: sarebbe interessante incorporare il criterio di selezione in un sistema intelligente "di sostegno" al giocatore parallelo (ad esempio un sistema esperto basato su regole). Il compito di tale agente è di individuare particolari caratteristiche della posizione corrente e in funzione di esse attribuire peso differente alle proposte delle istanze.

Il principale problema nell’indagine dei metodi descritti rimane tuttavia l’esplosione esponenziale del numero di differenti giocatori paralleli che possono scaturire in funzione di:

- numero di istanze
- attribuzione di conoscenza alle istanze
- attribuzione di pesi alle proposte delle diverse istanze.

Lo stesso problema si verifica, con minori gradi di libertà, nell’attribuzione ottimale di pesi ai parametri di una funzione di valutazione di un ordinario giocatore artificiale. Le soluzioni più interessanti a tale problema sono basate sulla regolazione automatica dei pesi (auto-tuning) ottenuta per approssimazioni successive della migliore soluzione. In generale questi metodi di ottimizzazione operano attraverso la ripetizione ciclica (fino al raggiungimento di una soluzione soddisfacente) delle seguenti fasi:

1. sono verificate, attraverso un database di posizioni di test, le prestazioni di certi giocatori artificiali originati da una certa attribuzione di pesi nella rispettiva funzione di valutazione

2. sulla base dei risultati della fase 1. sono determinate nuove attribuzioni di pesi (e quindi nuovi giocatori) che sintetizzano le principali qualità manifestate dai giocatori precedenti; tale sintesi può essere attuata con:

- metodi analitico-statistici [Ana90]
- algoritmi genetici [Ped91]
- reti neurali [vTi91]

3. torna alla fase 1. sperimentando i giocatori ottenuti nella fase 2.

È ragionevole l’utilizzo di questi metodi di indagine nello studio della distribuzione della conoscenza?

È completamente da indagare, infine, la possibilità di sviluppo di algoritmi ed euristiche di ricerca strettamente dipendenti dalle categorie di conoscenza usate nella funzione di valutazione. Ad esempio nell’esperimento Scouts [Sch87] è utilizzata l’euristica "null move" per aumentare

enormemente la profondità di una ricerca $\alpha\beta$ che valuta solo il materiale. La nostra idea è che si possa aprire un grande spazio di ricerca riguardo euristiche di ricerca specifiche, che hanno a che fare con funzioni di valutazione particolari, in contrapposizione ad euristiche di ricerca indipendenti dal dominio o che fanno riferimento a funzioni di valutazione di tipo generale (come sono quelle attualmente presenti nei giocatori artificiali più forti).

Riguardo il problema della distribuzione della conoscenza dirigente devono essere ancora verificate da un'analisi più attenta e dalla sperimentazione le idee proposte in 6.4, fondate su base assolutamente intuitiva.

È possibile confluire in un unico giocatore parallelo sia il concetto di distribuzione della ricerca che della conoscenza?

Si potrebbe pensare, ad esempio, ad un giocatore parallelo con due livelli logici di distribuzione:

• a più alto livello vi sono giocatori indipendenti con diversa conoscenza terminale i quali cooperano dopo il completamento delle rispettive ricerche selezionando (attraverso un criterio di selezione) una delle mosse da essi proposte

• tali giocatori sono in realtà dotati di ricerca distribuita (distribuzione a basso livello) la quale garantisce maggiore efficienza nella visita e quindi l'esplorazione più in profondità dell'albero di gioco.

Soluzione alternativa potrebbe essere l'applicazione in cascata delle due forme di distribuzione:

• nella prima fase tutte le risorse di elaborazione sono impegnate nella visita separata dell'albero di gioco disponendo ciascuna di differente conoscenza terminale

• nello stadio successivo il criterio di selezione diviene in realtà l'esplorazione delle sole mosse proposte nella prima fase; essa è attuata attraverso l'impiego di tutte le risorse di elaborazione guidato da un algoritmo di distribuzione della ricerca: la mossa che emerge migliore da questa ricerca (selettiva) sarà quella realmente proposta dal giocatore parallelo complessivo.

I due esempi appena indicati suggeriscono che la combinazione di distribuzione della ricerca e della conoscenza possa essere stabilita in modo molto naturale. Riteniamo che questo lavoro abbia fornito sufficienti indicazioni riguardo la possibilità di un "incontro" fra le due forme di distribuzione giustificando così future ricerche nella direzione appena tracciata di una loro combinazione.

Bibliografia

[Abr89] B. Abramson, "Control Strategies for Two-player Games", *ACM Computing Surveys*, Vol.21, No.2, (Jun.1989),137-161.

[AkBaDo82] S. Akl, D. Barnard, R. Doran, "Design, Analysis, and Implementation of a Parallel Tree Search Algorithm", *IEEE Transactions on Pattern Analysis and Machine Intelligence,* Vol. PAMI-4, No.2, (Mar.1982), 192-203.

[Alt89] I. Althöfer, "A Summary of some Results in theoretical game tree search and dreihirn-experiment", *Proc. Workshop on New Directions in Game-Tree Search*, Ed. T.A. Marsland, Edmonton, (1989), 16-32.

[Alt91] I. Althöfer, "Selective trees and majority systems: two experiments with commercial chess computers", Advances in computer chess 6, Ed. D. Beal, (1991), 37-59.

[Ana90] T. Anantharaman, "A statistical study of selective min-max search in computer chess", Carnegie-Mellon University, Pittsburgh, (May 1990).

[AsCaGe89] C. Ashcraft, N. Carriero, D. Gelernter, "Is explicit parallelism natural? Hybrid DB search and sparse LDL^T factorization using Linda", Yale University, New Haven, (Jan. 1989).

[BaKaLe92] H. Bal, M. Kaashoek, W. Levelt, "A comparison of two paradigms for distributed shared memory", Dept. of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, (1992).

[BaKaTa92] H. Bal, M. Kaashoek, S. Tanenbaum, "Orca: a language for parallel programming of distributed systems", *IEEE Transactions on Software Engineering*, Vol. 18, No. 3, (Mar. 1992), 190-205.

[Bal91] H. Bal, "Heuristic search in PARLOG using replicated worker style parallelism", *Future Generation Computer Systems*, North-Holland, 6, (1991), 303-315.

[Bal92] H. Bal, "A comparative study of five parallel programming languages", *Future Generation Computer Systems*, North-Holland, 8, (1992), 121-135.

[Bal90] H. Bal, "Programming distributed systems", Prentice Hall, Silicon Press, (1990).

[BalvRe86] H.E. Bal, R. van Renesse, "A summary of parallel alpha-beta search results", *ICCA Journal*, (Sep.1986), 146-149.

[Bau78a] G. Baudet, "The design and analysis of algorithms for asynchronous multiprocessors", Ph.D. dissertation, Computer Science Dept., Carnegie-Mellon University, Pittsburgh, (Apr.1978).

[Bau78b] G. Baudet, "On the branching factor of the alpha-beta pruning algorithm", *Artificial Intelligence*, Vol. 10, (1978), 173-199.

[Ber73] H. Berliner, "Some necessary conditions for a master chess program", *Proceedings of the Third Intenational Joint Conference on Artificial Intelligence*, Stanford, California, (Aug.1973), 77-85.

[Ber82] H. Berliner, "Search vs. knowledge: an analysis from the domain of games", Department of Computer Science, Carnegie-Mellon University, (1982).

[BerEbe86] H. Berliner, C. Ebeling, "The SUPREM architecture: a new intelligent paradigm", *Artificial Intelligence*, Vol. 28, (1986), 3-8.

[BjCaGe89] R. Byornson, N. Carriero, D. Gelernter, "The implementation and performance of hypercube Linda", Report RR-690, Yale University, New Haven, (Mar. 1989).

[BraKop82] I. Bratko, D. Kopec, "The Bratko-Kopec experiment: a comparison of human and computer performance in chess", *Advances in Computer Chess 3*, Ed. M.R.B. Clarke, Pergamon Press, (1982), 57-72.

[Bru63] A.L. Brudno, "Bounds and valuations for abridging the search of estimates", *Problems of Cybernetics*, Pergamon Press, Elmsford, N.Y., (1963), 225-241.

[CCCG85] N. Carriero, S. Chandran, S. Chang, D. Gelernter, "Parallel programming in Linda", *IEEE Parallel Processing Conf.*, (1985).

[CaGeLe88] N. Carriero, D. Gelernter, J. Leichter, "Distributed data structures in Linda", *Proceedings of the ACM Symposium on Principles of Programming Languages*, (Jan. 1986), 13-15.

[CarGel88a] N. Carriero, D. Gelernter, "Applications experience with Linda", Yale University, New Haven, (Jan. 1988).

[CarGel88b] N. Carriero, D. Gelernter, "How to write parallel programs: A guide to the perplexed", Yale University, New Haven, (Nov. 1988).

[CarGel89] N. Carriero, D. Gelernter, "Linda in context", *Communications of the ACM*, (Apr. 1989), 444-458.

[CarGel90] N. Carriero, D. Gelernter, "How to write parallel programs. A first course", The MIT Press, (1990).

[Cia92] P. Ciancarini, "I giocatori artificiali di scacchi", Ed. Mursia, (1992).

[CiaGas89] P. Ciancarini, M. Gaspari, "A knowledge-based system and a developement interface for the middle game in chess", *Advances in Computer Chess 5*, D.F Beal (Editor), (1989).

[CFC84] CFC, Chess federation of Canada Rating System, *En Passant*, (1984), 63-64.

[DanMah93] C. Daniels, A. Mahanti, "A SIMD approach to parallel heuristic search", *Artificial Intelligence*, Vol. 60, (1993), 243-282.

[dGr65] A.D. de Groot, "Thought and Choice in Chess", Mouton, The Hague, (1965).

[Ebe86] C. Ebeling, "All the Right Moves: A VLSI Architecture for Chess", The MIT Press, (1986).

[EdwHar63] D. Edwards, T. Hart, "The alpha-beta heuristic", Tech. Rep. 30, MIT AI Memo, Computer Science Dept., Massachusetts Institute of Technology, Cambridge, (1963).

[Eli90] R.J. Elias, "Oracol, A Chess Problem Solver in Orca", Dept. of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, (1990)

[FaFeGe91] M. Factor, S. Fertig, D. Gelernter, "Using Linda to build parallel AI applications", Yale University, New Haven, (Jun. 1991).

[FacGel88] M. Factor, D. Gelernter, "The parallel process lattice as an organizing scheme for realtime knowledge daemons", Yale University, New Haven, (Mar. 1988).

[FerGel91] S. Fertig, D. Gelernter, "The design, implementation, and performance of a database-driven expert system", Yale University, New Haven, (Feb. 1991).

[FeKoPo93] C. Ferguson, R. Korf, C. Powley, "Depth-first heuristic search on a SIMD machine", *Artificial Intelligence*, Vol. 60, (1993), 199-242.

[FeMyMo91] R. Feldmann, P. Mysliwietz, B. Monien, "A Fully Distributed Chess Program", Advances in computer chess 6, , Ed. D. Beal, (1991), 1-27.

[FeMyMo92] R. Feldmann, P. Mysliwietz, B. Monien, "Experiments with a Fully-Distributed Chess Program", *Heuristic Programming in AI*, Eds. J. Van Den Herik e V. Allis, Hellis Horwood, (1992), 72-87.

[FelOtt88] E.W. Felten, S.W. Otto, "A Highly Parallel Chess Program", *Proceedings of the International Conference on Fifth Generation Computer Systems*, (1988), 1001-1009.

[FisFin80] J.P. Fishburn, R.A. Finkel, "Parallelism in Alpha-Beta Search on Arachne", Technical Report 394, Computer Science Dept., University of Wisconsin, Madison, (Jul.1980).

[FinFis82] J.P. Fishburn, R.A. Finkel, "Parallelism in alpha-beta search", *Artificial Intelligence*, Vol. 19, (1982), 89-106.

[FMMV89] R. Feldmann, P. Mysliwietz, B. Monien, O. Vornberger, "Distributed Game-Tree Search", *ICCA Journal*, Vol.12, No.2, (1989), 65-73.

[Gel85] D. Gelernter, "Generative communication in Linda", *ACM Trans. Progr. Lang. Syst. 7*, 1 (1985), 80-112

[GelKam92] D. Gelernter, D. Kaminsky, "Supercomputing out of recycled garbage: preliminary experience with Piranha", *Proceedings of the ACM Int. Conf. on Supercomputing*, (Jul. 1992).

[Gil72] J. Gillogly, "The technology chess program", *Artificial Intelligence*, Vol. 3, (1972), 145-163.

[Gil78] J. Gillogly, "Performance analysis of the technology chess program", Carnegie Mellon University, Pittsburgh, (Mar. 1978).

[HLMSW92] F. Hsu, R. Levinson, J. Schaeffer, T.A. Marsland, D. Wilkins, "Panel: the role of chess in artificial intelligence research", D.E in *Proc. 12th Int'1 Joint Conference in Artificial Intelligence*, (1992), 547-552.

[Hsu90] F. Hsu, "Large scale parallelization of alpha-beta search: an algorithmic and architectural study with computer chess", Ph.D. thesis, Carnegie Mellon University, Pittsburgh, (Feb. 1990).

[KerRit78] B. Kernighan, D. Ritchie, "The C programming language", Prentice-Hall, (1978).

[KnuMoo75] D.E. Knuth, R.W. Moore, "An analysis of alpha-beta pruning", *Artificial Intelligence*, Vol.6, (1975), 293-326.

[Kor85] R. Korf, "Depth-first iterative-deepening: An optimal admissible tree search", *Artificial Intelligence*, Vol. 27, (1985), 97-109.

[Kor90] R. Korf, "Real-time heuristic search", *Artificial Intelligence*, Vol. 42, (1990), 189-211.

[Kor93] R. Korf, "Linear-space best-first search", *Artificial Intelligence*, Vol. 62, (1993), 41-78.

[LanSmi93] K. Lang, W Smith, "A test suite for chess programs", Draft, (Jun. 1993).

[LeiWhi88] J. Leichter, R. Whiteside, "Using Linda for supercomputing on a local area network", Yale University, New Haven, (Jun. 1986).

[Lin90] "C-Linda Reference manual", Scientific computing associates inc., New Haven, Connecticut, (1990).

[Low89] C. Low, "Parallel game tree searching with lower and upper bounds", University of Illinois at Urbana-Champaign, (1989).

[MaBrSu91] T.A. Marsland, T. Breitkreutz, S. Sutphen, "A network multi-processor for experiments in parallelism", *Concurrency: Practice and Experience*, Vol.3, No.3, (Jun.1991), 203-219.

[MarCam82] T.A. Marsland, M. Campbell, "Parallel Search of Strongly Ordered Game Trees", *ACM Computing Surveys*, Vol.14, No.4, (Dec.1982), 533-551.

[MaOlSc86] T.A. Marsland, M. Olafsson, J. Schaeffer, "Multiprocessor tree-search experiments", *Adavances in Computer Chess 4*, Pergamon Press, (1986), 37-51.

[MarPop83a] T.A. Marsland, F. Popowich, "A multiprocessor tree-searching system design", University of Alberta, Edmonton, Canada, (Jul. 1983).

[MarPop83b] T.A. Marsland, F. Popowich, "Experience with a parallel chess program", University of Alberta, Edmonton, Canada, (Aug. 1983).

[MarPop85] T.A. Marsland, F. Popowich, "Parallel Game-Tree Search", *IEEE Transactions on Pattern Analysis and Machine Intelligence,* Vol.7, No.4, (Jul.1985), 442-452.

[McA85] D.A. McAllester, "A New Procedure for Growing Mini-Max Trees", Technical Report, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, (1985).

[Nau82] D.S. Nau, "The last player theorem", *Artificial Intelligence*, Vol.18, (1982), 53-65.

[New79] M. Newborn, "Recent progress in computer chess", *Advances in Computers*, Academic Press, (1979), 59-117.

[New88] M. Newborn, "Unsynchronized Iteratively Deepening Parallel Alpha-Beta Search", *IEEE Transactions on Pattern Analysis and Machine Intelligence,* Vol.10, No.5, (Sep.1988), 687-694.

[Pea80] J. Pearl, "Asymptotic Properties of Minimax Trees and Game-Searching Procedures", *Artificial Intelligence*, Vol.14, No.2, (1980), 113-138.

[Pea82] J. Pearl, "The solution for the branching factor of the alpha-beta pruning algorithm and its optimality", *Comm. of the ACM 25*, 8, (1982), 559-564.

[Riv88] R. Rivest, "Game tree searching by min/max approximation", *Artificial Intelligence*, Vol. 34, (1988), 77-96.

[Ros81] P.S. Rosenbloom, "A world-championship-level othello program", CMU-CS-81-137, Department of Computer Science, Carnegie-Mellon University, (1981).

[RotSet93] R. Roth, T. Setz, "LIPS: a system for distributed processing on workstations", Draft, (Feb. 1993).

[Sam60] A. L. Samuel, "Programming computers to play games", *Advances in Computers*, Vol.1, (1960), 165-192.

[Sha50] C.E. Shannon, "Programming a computer for playing chess", *Philosophical Magazine* , Vol.41, (1950), 256-275.

[Sch86] J. Schaeffer, "Experiments in search and knowledge", University of Alberta, Edmonton, Canada, (Jul. 1986).

[Sch87] J. Schaeffer, "Experiments in distributed game-tree searching", University of Alberta, Edmonton, Canada, (Jan. 1987).

[Sch89] J. Schaeffer, "Distributed Game-Tree Searching", *Journal of Parallel and Distributed Computing*, Vol.6, (1989), 90-114.

[Sch90] J. Schaeffer, "Conspiracy numbers", *Artificial Intelligence*, Vol. 43, (1990), 67-84.

[SlaDix69] J.R. Slagle, J.K. Dixon, "Experiments with some programs that search trees", *J. ACM*, Vol.16, No.2, (Apr.1969), 189-207.

[SlaAtk77] D.J. Slate, L.R. Atkin, "Chess 4.5-the Northwestern University chess program, *Chess Skill in Man and Machine*, P.W. Frey, Ed. Springer-Verlag, New York, (1977), 82-118.

[StaWal88] C. Stanfill, D. Waltz, "Artificial intelligence related search on the connection machine", *Proc. of the Int. Conf. on Fifth Generation Computer Systems*, (1988).

[Sti89] L. Stiller, "Parallel Analysis of Certain Endgames", *ICCA Journal*, Vol. 12:2, (1989), 55-64.

[Sti91] L. Stiller, "Group Graphs and Computational Symmetry on Massively Parallel Architecture", *The Journal of Supercomputing*, Vol.5, (1991), 99-117.

[Tho82] K.Thompson, "Computer Chess Strength", *Advances in Computer Chess 3*, Ed. M.R.B. Clarke, Pergamon Press, (1982), 55-56.

[Thu92] F. Thuijsman, "An introduction to the theory of games", *Heuristic Programming in AI*, Eds. J. Van Den Herik e V. Allis, Hellis Horwood, (1992), 205-220.

[Tun91] W. Tunstall-Pedoe, "Genetic algorithms optimizing evaluation functions",*ICCA Journal*, Vol. 14, No. 3, (Sep. 1991), 119-128.

[vNeMor44] J. von Neumann, O. Morgenstern, "Theory of Games and Economic Behavior", Princeton Univ. Press, Princeton, N.J., (1944).

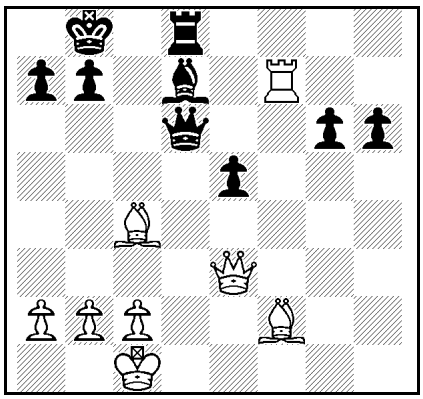
[vTi91] A. van Tiggeken, "Neural networks as a guide to optimization. The chess middle game explored", *ICCA Journal*, Vol. 14, No. 3, (Sep. 1991), 115-118.

[Wil79] D.E. Wilkins, "Using patterns and plans to solve problems and control search", Ph.D. thesis, Department of computer science, Stanford University, (1979).

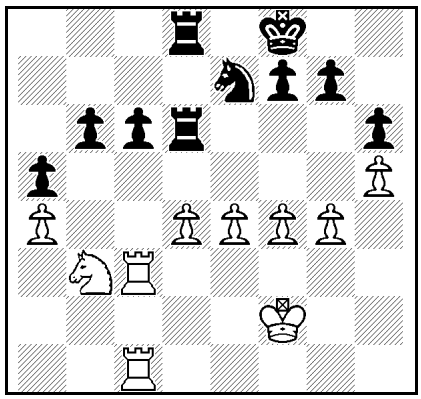
[Wil87] W. Wilson, "Concurrent alpha-beta. A study in concurrent logic programming", *Int. Conf. on Logic programming*, S. Francisco, (1987).

[Zen90] S.E. Zenith, "Linda coordination language: Subsystem kernel architecture (on transputers), RR-794, Yale University, New haven, (May 1990).

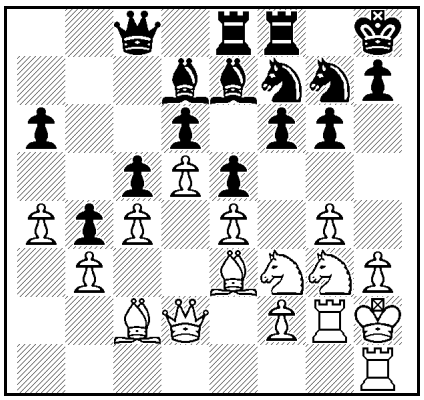
Appendice A
Posizioni di Bratko-Kopec



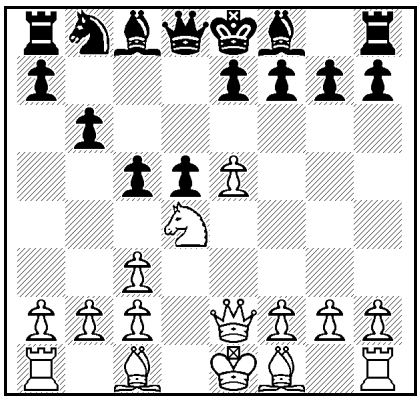
1. ♕d6-d1



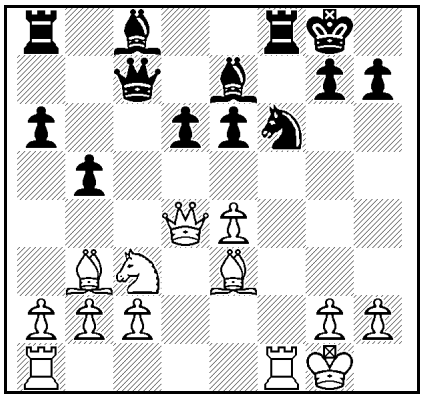
2. ♙d4-d5



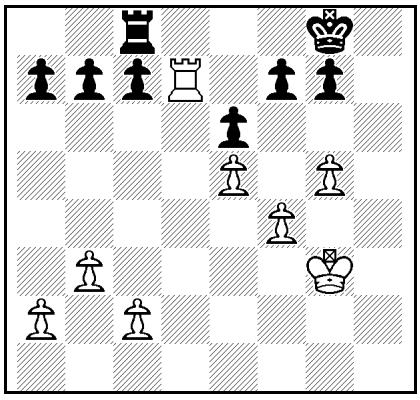
3. ♟f6-f5



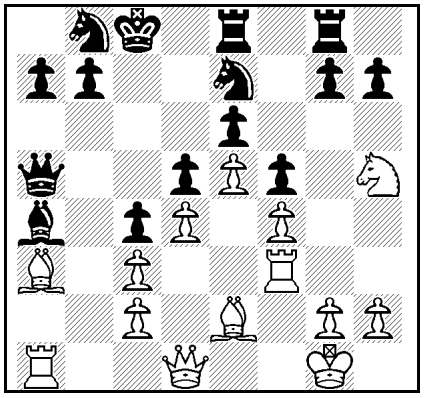
4. ♙e5-e6



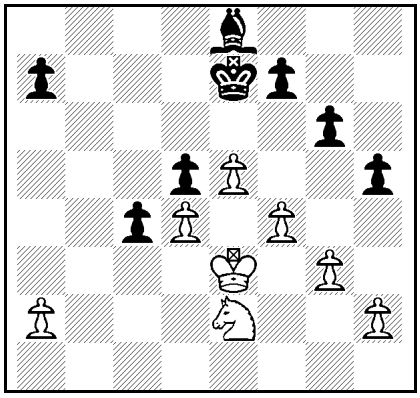
5. ♙a2-a4 / ♘c3-d5



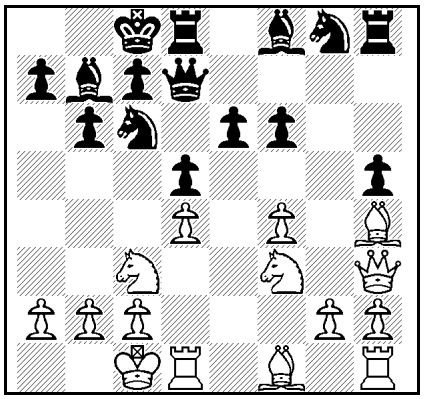
6. ♙g5-g6



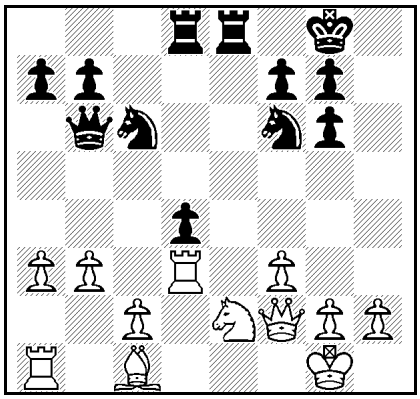
7. ♘h5-f6



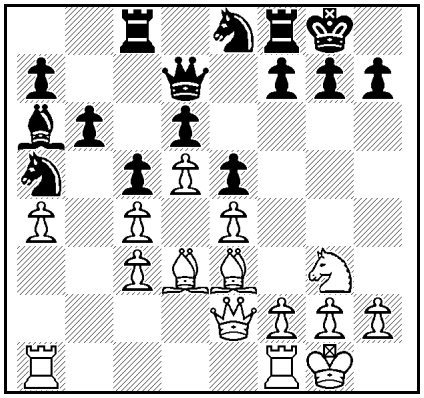
8. ♙f4-f5



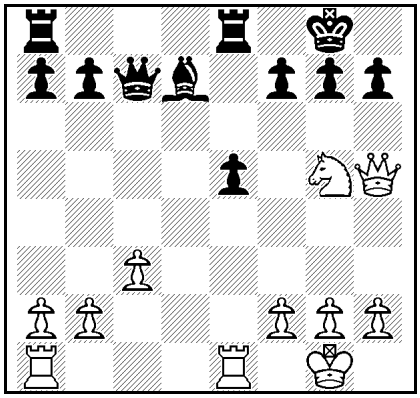
9. ♙f4-f5



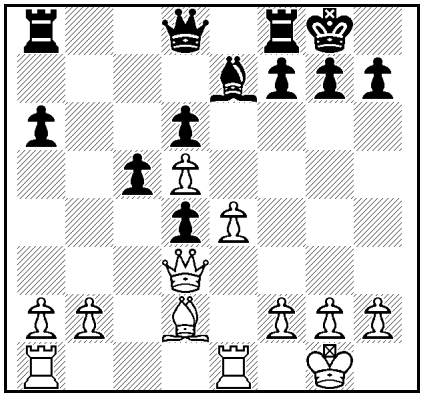
10. ♞c6-e5



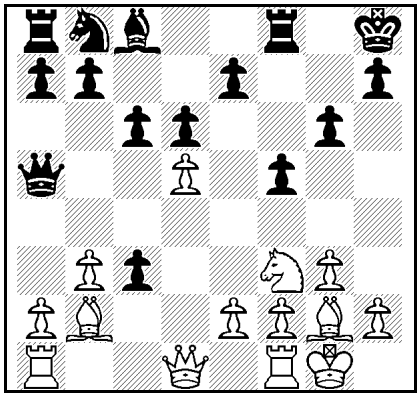
11. ♙f2-f4



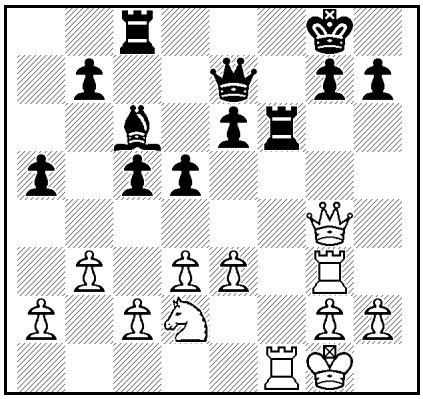
12. ♝d7-f5



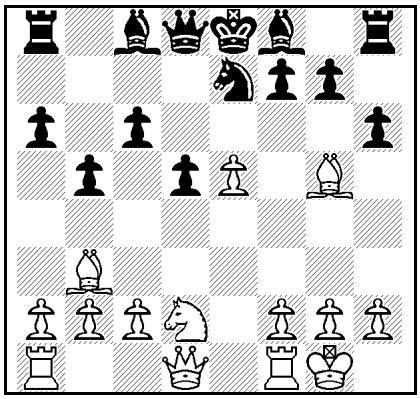
13. ♙b2-b4



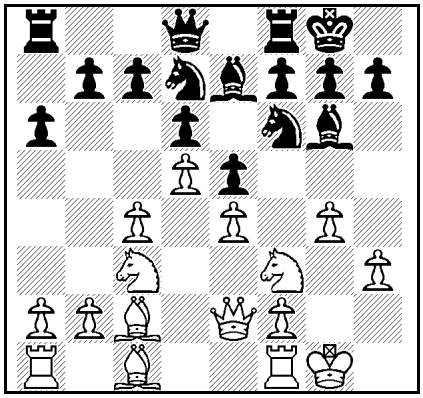
14. ♕d1-d2 / ♕d1-e1



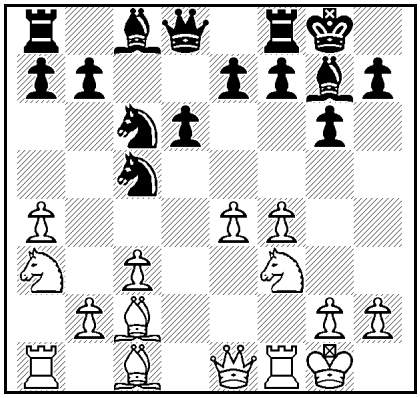
15. ♕g4x♟g7



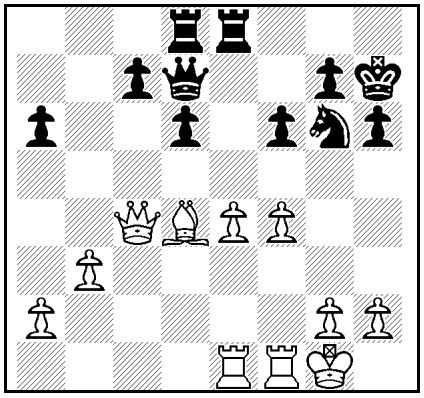
16. ♘d2-e4



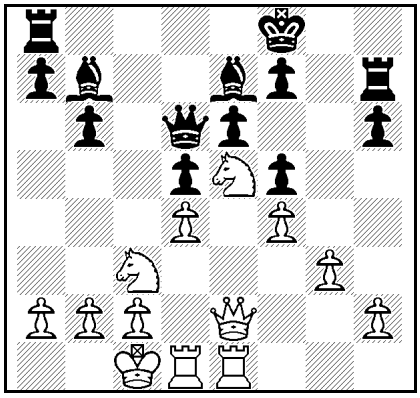
17. ♟h7-h5



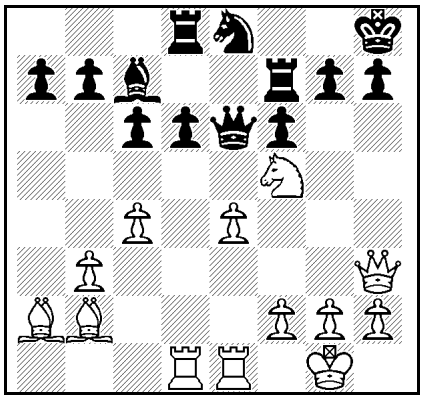
18. ♘c5-b3



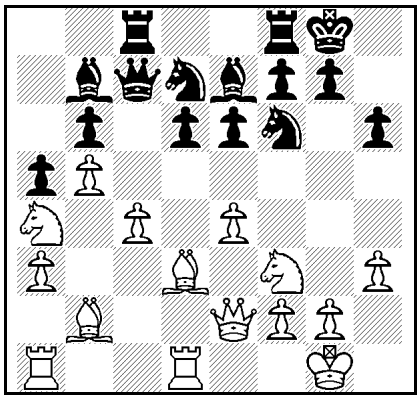
19. ♖e8x♙e4



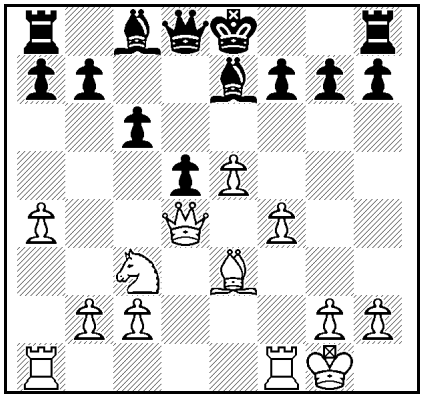
20. ♙g3-g4



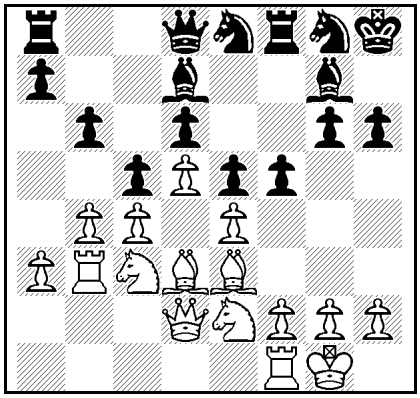
21. ♘f5-h6



22. ♗b7x♙e4



23. ♟f7-f6



24. ♙f2-f4