Università degli studi di Bologna

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI
Corso di Laurea in Informatica
Materia di Tesi: Laboratorio di Ingegneria del Software

PROGETTO E REALIZZAZIONE DI UN PROGRAMMA CLIENTE PER INTERNET CHESS CLUB

Tesi di Laurea di:
MARCO COLLAREDA

Relatore:
Prof. PAOLO CIANCARINI

Parole chiave: servizi Internet, client, server, scacchi, Internet Chess Server

Anno Accademico 2001-2002
Sessione autunnale

Indice

ABSTRACT

Chess has been for more than fifty years one of the most interesting game for the computer scientists. Much work has been done over the years and all of the researchers in the game theory field have, sooner or later, applied their algorithms to chess.

Studies about chess programs became concrete during the post-war period, thanks to the contribution of scientists such as Claude Shannon and Alan Turing. From then on chess engines improved a lot, until in 1997 the most powerful chess program managed to defeat the World Champion.

From the computer science point of view chess is such an interesting game because its application on a computer is a trade-off between purely mechanical algorithms and hardware limitations. For this reason a chess program must use some methods to override such obstacle and this is what involved the researchers in Artificial Intelligence field.

Recently, as computer technology evolved, we have seen the diffusion of many programs, even pretty advanced, applied to chess. This is not true only for chess engines, but for graphical interfaces and several kinds of tools as well.

The aim of this paper is, first of all, to show the structure of a chess engine; we will illustrate the principles of the most used algorithms, of some data structures used to make programs be faster and faster, and of all those techniques which over the years have been frequently used in most of chess programs.

One of our purposes is to analyse two existing chess engines, showing their architecture, the algorithms and the data structures they are made up of, play strategies, the methods and the techniques they implement.

Thereafter, we will see how the interest has moved from chess engines to applications. On the net, in fact, there are lots of programs that implement almost any service connected with chess we can think about. We can find chess games viewers and editors, games or players database managements, graphical interfaces to play chess with a user-friendly board and much more. We are going to dwell on graphical user interfaces and we are going to analyse one of the most popular software used by chess players on the net.

We will also see how the new frontier has moved from desktop to the net, thanks to the larger and larger spreading of Internet Chess Servers. They are clubs that involve thousands of users that play chess on net. Even for this topic we will give an example by describing the features of the largest and most visited Internet Chess Server.

Even the software developed by the author of this paper deals with Internet Chess Servers. It has to do with saving games stored on the server. This purpose has been achieved by connecting to an Internet Chess Server, sending the needful commands and reading the output according to the rules of output protocol.

Let’s see then, chapter by chapter, the structure of such plan.

In chapter one we understand why chess is an interesting trial for who is keen on game theory. After a brief outline of chess engines history we see the main technical characteristics of Deep Blue, the IBM supercomputer that in 1997 defeated the World Champion Garry Kasparov. Finally we make a contribution for the old matter that has been discussed by the researchers for many years: are chess engines intelligent programs?

Chapter two is meant to describe chess programs structure. After explaining the principles of game theory, that is the algorithms that base themselves upon trees of moves, we see the typical features of a chess engine, such as board representation, moves generation, as well as all those techniques improved over the past decades that are used to make engines more and more fast and effective.

In chapters three and four we get the heart of chess programs by looking into two open source chess engines from a more detailed point of view: GNU Chess and Crafty. Our purpose is to describe the algorithms they use, the data structures they avail themselves of, how they implement the techniques we have introduced in chapter two and which service and interface they offer to the user.

Chapter five is a short introduction to the graphical interfaces, that is those programs that let the human interact with the chess program in a more faster and comfortable way.

In chapter six an example of graphical interface is shown: Winboard/Xboard interface for Windows and Linux environments. We describe its main features and the services it offers. We see then, quite in detail, how the interface interacts with the chess engine below and how it communicates with the Internet Chess Server.

Chapter seven takes the reader through the Internet Chess Servers, by describing their characteristics and showing all that a user can do by connecting to a chess server.

The aim of chapter eight is to show an Internet Chess Server in detail: Internet Chess Club. We see how we can connect, play a game, communicate, chat etc. For each service available that we are going to analyse we show the necessary commands, trying to show how is possible to achieve the same goal by inserting the command by hand or by using a graphical interface. We then go down and reach a more detailed level: we examine the communication protocol that the server uses to swap data with the client.

In chapter nine we present PGN-Saver, the software developed by the author of this paper to show how a program can communicate with an Internet Chess Server. Its features and the commands that the user can insert are fully described. After that we will explain how the program works: the commands it sends to the server and how it treats data sent back from the server. We will also explain how the program reads the output of the server and minds its protocol rules.

Finally, in chapter ten, we briefly describe all the other kinds of tools that can be found on the Web; furthermore we add for every type some examples and the address from where you can download a copy.

After a short conclusion about the work done and the possible themes that we could face to in the future, in appendix you could find the source code of PGN-Saver.

1 Introduzione

1.1 Perché gli scacchi?

Perché molti informatici hanno impiegato tempo ed energie per sviluppare programmi per scacchi? Perché ancora oggi c'è qualcuno che si preoccupa di rendere i programmi per scacchi sempre più veloci, più efficienti, più efficaci? Perché l'applicazione al calcolatore di altri tipi di giochi ha avuto poca fortuna, mentre dopo cinquant'anni gli scacchi sono ancora sulla bocca di ogni informatico appassionato di teoria dei giochi?

Innanzitutto i giochi a informazione completa[1] sono stati i primi ad essere studiati ed applicati all'informatica, proprio perché l'informazione completa è un requisito fondamentale per usare certi algoritmi di ricerca esaustiva. I giochi ad informazione incompleta, invece, sono meno trattati dei primi: richiedono infatti analisi particolari, basi di conoscenza, regole inferenziali; i programmi che li realizzano cercano in sostanza di imitare i processi deduttivi dell'essere umano e per questo sono molto più complessi. Gli scacchi sono un gioco a informazione completa.

Mentre un gioco a informazione completa piuttosto banale (ad esempio il filetto o il forza quattro) viene facilmente risolto da un programma usando determinati algoritmi[2], gli scacchi hanno una caratteristica che li distingue: il numero di combinazioni di mosse possibili è elevatissimo, e per questo è impensabile che un

[1] *Gioco a informazione completa*: gioco in cui due giocatori effettuano alternativamente una mossa avendo in ogni istante a) tutte le informazioni sullo stato del gioco, b) le stesse informazioni dell'avversario.
[2] Vedi cap. 2.

calcolatore riesca ad effettuare una ricerca esaustiva nell’albero per trovare la mossa ottimale.

In una partita di scacchi, ad ogni turno, le mosse possibili sono in media circa venticinque. In una partita media ci sono circa 40 mosse, dove per mossa si intende una semimossa (ply) bianca e una semimossa nera. Il numero di partite possibili, quindi, è di 25 elevato alla 80: un numero enorme. Anche restringendo il campo alle sole mosse sensate, che è in media di quattro o cinque sulle 25 totali, il numero di combinazioni possibili si abbassa a cinque elevato alla 80. Alex G.Bell, uno studioso di intelligenza artificiale, ha calcolato che se anche esistesse una macchina in grado di giocare una partita in un nanosecondo, il che significa un miliardo di partite al secondo, e se anche un milione di tali macchine lavorasse a tempo pieno dal momento della nascita del sistema solare, solo un milionesimo delle partite sensate possibili potrebbe essere esaminato!

Questo ci fa capire 'perché gli scacchi'. Sono un gioco ad informazione completa, cioè teoricamente risolvibile in modo meccanico attraverso semplici algoritmi, ma non risolvibile nella realtà in un tempo finito. Per questo il programmatore è costretto a trovare compromessi, strategie, tecniche, o addirittura nuove tecnologie, per rendere i programmi quanto più veloci ed efficienti possibile.

1.2 Breve storia dei motori scacchistici

Già il matematico inglese Charles Babbage dopo aver progettato la macchina analitica volle dimostrare che essa era in grado di realizzare 'cose intelligenti' scegliendo come banco di prova gli scacchi e indicando le regole che la macchina avrebbe dovuto seguire per giocare la partita.

Nel 1890 le idee di Babbage trovarono una prima applicazione pratica nella macchina dello spagnolo Leonardo Torres y Quevedo, capace di giocare un semplice finale di Re e Torre contro Re solo, muovendo i pezzi bianchi fino ad arrivare allo scacco matto.

I primi veri studi teorici sui motori scacchistici non furono effettuati prima della Seconda Guerra Mondiale. Nel 1946 Alan Turing ideò un complesso algoritmo in cui fu usato un metodo riconducibile alle odierne funzioni di valutazione (calcolo del

materiale secondo dei valori convenzionali, mobilità dei pezzi)[3]. La prima partita fra un giocatore artificiale e un essere umano fu giocata a Manchester nel 1952 e vide opposti il programma di Turing e Alick Glennie. Non fu propriamente il programma di Turing a giocare la partita, bensì Turing stesso, che, non avendo a disposizione un calcolatore adatto, si armò di carta e penna ed eseguì ogni singola istruzione del programma fino alla determinazione della mossa ottimale. La partita finì con la vittoria di Glennie per abbandono dell'avversario.

Nel 1949 l'inglese Claude Shannon, riprendendo gli studi di Turing, descrisse come doveva operare un programma per giocare correttamente a scacchi. Il suo lavoro, puramente teorico, ispirò Alex Bernstein, che nel 1950 implementò un programma di scacchi su un elaboratore IBM 704.
Tale programma sceglieva ogni volta tra sette diverse possibilità, calcolando per ciascuna sette risposte, poi ancora sette risposte per ogni risposta e quindi altre sette risposte. In totale sette elevato alla quarta potenza, cioè 2401 mosse diverse ogni volta. Per effettuare una mossa l'elaboratore impiegava circa otto minuti ed il suo livello di gioco era pari a quello di un modesto principiante.

È datata 1965 la prima sconfitta di un essere umano da parte di un giocatore artificiale. L'uomo era il professor Hubert Dreyfus e il programma il *Machack*, sviluppato presso il Tecnology Institute del Massachusetts; Machack girava su un Digital PDP-6.

Questo programma è diventato famoso anche per un motivo ulteriore. Nel 1967 gli venne sottoposta una posizione giudicata allora come 'pari', dato che non si riusciva a trovare una continuazione vincente per il nero, cui toccava muovere. Il programma, invece, scoprì una brillante soluzione, basata sul sacrificio di una torre, che portava rapidamente allo scacco matto.

Nel 1968 Edimburgo ospitò la prima conferenza internazionale sui programmi di scacchi per computer. In quell'occasione, il maestro internazionale David Levy scosse l'ambiente con una singolare scommessa: scommise 250 sterline, somma che poi venne aumentata a 1250 sterline per l'intervento di altri scommettitori, che nei successivi dieci anni nessun programma sarebbe riuscito a batterlo. La sfida fu raccolta da molti appassionati e diede un grande impulso alla ricerca.

L'interesse crebbe rapidamente. Nel 1974 si svolse a Stoccolma il primo campionato mondiale per programmi di scacchi, vinto da un programma sovietico di

[3] Per questi concetti vedi cap. 2.

nome *Kaissa*. I successivi campioni del mondo, motori scacchistici che sono rimasti nella storia, furono *Chess 4.6, Belle* e *Cray Blitz.* Nel 1980 venne organizzato il primo campionato del mondo per micro-computer e nel 1982 due giocatori artificiali parteciparono per la prima volta ad un torneo per esseri umani, con performance di tutto rispetto.

Nel 1989 David Levy, che aveva proposto la scommessa nel 1968, venne finalmente sconfitto da Deep Thought, un supercalcolatore della IBM. Ben presto si intuì che i calcolatori stavano cominciando a rovesciare il rapporto di forza. Nella primavera del 1994 un programma chiamato *Fritz* arrivò primo in un torneo lampo[4], alla pari di Kasparov, riuscendo addirittura a batterlo in una partita lampo. Da sottolineare il fatto che Fritz non era eseguito su un supercalcolatore, ma su un normale Personal Computer.

Nel 1996 l'IBM propose a Kasparov una sfida contro il suo nuovo supercalcolatore scacchistico, *Deep Blue*, evoluzione di Deep Thought, sulla base di sei partite, con tempi di riflessione regolamentari. Il campione del mondo in carica sconfisse il computer per quattro a due, ma perse clamorosamente la partita d'esordio: pertanto il 10 febbraio 1996 Deep Blue entrò nella storia come il primo computer ad aver battuto un campione mondiale di scacchi in una partita giocata su tempi regolamentari.

L'anno successivo l'IBM propose una rivincita. Nella versione precedente Deep Blue era in grado di calcolare 280 milioni di mosse al secondo, ma non applicava particolari criteri di selezione delle mosse se non quello del guadagno di materiale. Nella versione '97 i tecnici dell'IBM decisero di far calcolare al computer solo 220 milioni di mosse al secondo, ma applicando algoritmi di gioco più sofisticati. Le ricerche informatiche infatti avevano dimostrato che la sola forza bruta di calcolo non è sempre sufficiente a far giocare bene una macchina, per quanto veloce essa sia. A questo si aggiunse il miglioramento dell'hardware del computer. La sfida terminò con due vittorie della macchina, una di Kasparov e tre patte: aveva vinto Deep Blue. Era l'11 maggio 1997.

Nel 1998 l'indiano Anand, vicecampione mondiale venne sconfitto dal programma *Rebel 10,* che girava su un computer basato sul microprocessore AMD K6-2 a 450 MHz, non certo un supercalcolatore paragonabile a quello usato per Deep Blue, e non certo in grado di calcolare 220 milioni di mosse al secondo.

[4] *Torneo lampo*: torneo in cui il giocatore è soggetto a forti vincoli di tempo per effettuare una mossa.

Negli ultimi anni i programmi per scacchi hanno fatto progressi molto notevoli. I programmi sono sempre più veloci, le barriere del numero di mosse valutate al secondo sono state abbattute una dopo l'altra. Nel 1996 Robert Hyatt lancia *Crafty*[5], si tratta di un progetto open source. Da quel momento Internet diventa un luogo dove i programmatori sviluppano programmi, comunicano, migliorano idee precedenti, si sfidano; l'elenco dei motori per scacchi open source scaricabili dalla rete è infinito.

1.3 Ma si tratta davvero di intelligenza artificiale?

Molti sono ancora oggi scettici sui programmi scacchistici. Si può veramente parlare di intelligenza artificiale quando un calcolatore elettronico sconfigge un essere umano al gioco? I detrattori sostengono che le macchine che giocano a scacchi si basano su mere ricerche di vasta portata, ricerche che non hanno nulla a che fare coi processi mentali di un campione di scacchi, come per esempio il riconoscimento di situazioni già affrontate, l'esclusione a priori di determinate mosse, la visione della scacchiera e il riconoscimento delle posizioni per porzioni e sottoinsiemi di pezzi, l'adozione di particolari strategie, magari dettate anche dalla situazione psicologica dell'avversario.

Al termine della sconfitta di Kasparov da parte di Deep Blue, il vicecampione mondiale Anand disse:

"per me Deep Blue non ha nulla a che fare con l'intelligenza artificiale. Esso rappresenta solamente l'incredibile violenza bruta che queste macchine esprimono, né più né meno".

I fondatori dell'Intelligenza Artificiale (termine coniato nel 1956) John McCarthy, Marvin Minsky, Allan Newell, Herbert Simon e Claude Shannon, ne diedero la seguente definizione:

l'Intelligenza Artificiale è una disciplina che studia la possibilità di ottenere dalle macchine prestazioni che, se compiute da un essere umano, sarebbero definite intelligenti.

[5] Vedi cap. 4.

Più tardi tale definizione venne rivista e arricchita.

L'Intelligenza Artificiale è quella disciplina, appartenente all'informatica, che studia i fondamenti teorici, le metodologie e le tecniche che permettono di progettare sistemi digitali (hardware) e sistemi di programmi (software) capaci di fornire all'elaboratore elettronico delle prestazioni che, ad un osservatore comune, sembrerebbero essere di pertinenza esclusiva dell'intelligenza umana.

La prima definizione afferma che l'AI concerne "*prestazioni che, se compiute da un essere umano, sarebbero definite intelligenti"*, la seconda "*prestazioni che, ad un osservatore comune, sembrerebbero essere di pertinenza esclusiva dell'intelligenza umana"*. Non c'è alcun assunto su come la macchina deve simulare questa intelligenza umana, anzi si fa proprio riferimento ad azioni che *se* compiute da un essere umano sarebbero definite come intelligenti, senza specificare quale sia la differenza tra l'intelligenza delle macchine e quella degli esseri umani.

D'altra parte sostenere che l'unica forma d'intelligenza sia quella umana è molto presuntuoso. Affermare che se una macchina esegue dei compiti apparentemente intelligenti usando determinate tecniche che non hanno niente a che vedere con i processi mentali umani significa equivocare la definizione di Intelligenza Artificiale.

Negli ultimi anni c'è chi si sta impegnando a realizzare motori scacchistici *intelligenti*, secondo l'accezione di intelligenza umana, usando regole inferenziali, insegnando al programma a imparare dai propri errori e adottando altre tecniche e algoritmi che tentano di imitare i processi propri del pensiero umano.

1.4 Deep Blue

Il progenitore di Deep Blue è un programma chiamato *Chiptest*, sviluppato da Feng-Hsiung Hsu nel 1985. Nel 1989 Hsu entrò a far parte dell'IBM e, con l'aiuto del collega Murray Campbell, si concentrò sullo studio dell'applicazione del calcolo parallelo per affrontare problemi di ricerca sempre più complessi. Ai due si unirono ben presto altri informatici, che si prefissero di produrre processori dedicati alla valutazione di mosse in una partita di scacchi.

L'ultima versione di Deep Blue utilizza un processore parallelo IBM RS/6000 SP a 32 nodi, formato da processori P2SC (Power Two Super Chip). Ognuno dei 32 nodi utilizza otto processori dedicati VLSI, per un totale di 256 processori che lavorano in parallelo. Il codice di Deep Blue è scritto in linguaggio C e lavora sotto il sistema operativo AIX. Ne esce un sistema scalabile, ad elevato parallelismo, in grado di calcolare fino a 50-100 miliardi di posizioni in tre minuti.

La velocità è senza dubbio il punto di forza di Deep Blue, e questo sembra essere il principale obiettivo dei suoi sviluppatori, convinti che la velocità sia la chiave della forza per un programma di scacchi. È stato calcolato che Deep Blue riesce a valutare oltre 200 milioni di mosse al secondo e che Kasparov arrivi a tre mosse al secondo.

La funzione di valutazione[6] di Deep Blue si basa su quattro fattori: materiale, posizione, sicurezza del re e tempo. Per esaminare le posizioni sulla scacchiera Deep Blue realizza un metodo detto *selective extensions*[7]. Invece di usare la cosiddetta forza bruta il programma seleziona via via dei percorsi da seguire, eliminando le mosse meno rilevanti.

1.5 Scacchi e computer

Chi è appassionato di scacchi deve considerarsi fortunato: in rete esiste una tale quantità di programmi e tool, molti dei quali gratuiti, da perdere la testa.

Oltre ai motori scacchistici, la maggior parte dei quali sono open source e quindi liberamente scaricabili, esistono siti che mettono a disposizione molte interfacce grafiche[8], alcune complete anche di motore. Esistono poi numerosi tool di varia natura: per personalizzare interfacce grafiche esistenti, per corredarle dei fonts, i

[6] Vedi sez. 2.4.
[7] Vedi sez. 2.2.11.
[8] Vedi cap. 5.

colori e le immagini preferite, e ancora utility per la gestione di file PGN o EPD[9], database per l'archiviazione di partite, e altro ancora.

Il gioco degli scacchi ha ricevuto una spinta dopo l'avvento di Internet. In rete si sono venute a formare vere e proprie comunità di giocatori, che, oltre a comunicare, scambiarsi idee e opinioni attraverso gruppi di discussione e mailing list, organizzano veri e propri tornei telematici, sia per esseri umani che per programmi. Sono nati poi molti Internet Chess Server[10], a cui i giocatori si collegano con la possibilità di sfidare esseri umani o programmi. Questi server mettono a disposizione molti altri servizi, ad esempio offrono la possibilità di osservare dal proprio PC una partita giocata da due maestri di scacchi.

Tra il software esistente in rete ci sono anche parecchi client per Internet Chess Server, corredati di scacchiera elettronica, che permettono, attraverso un unico programma, di effettuare il login al server, cercare giocatori da sfidare, giocare partite, ma anche salvare partite giocate; offrono un'ampia gamma di servizi attraverso una gradevole interfaccia grafica, per sopperire alla scomodità di inserire i comandi da console se si accede ad un Internet Chess Server senza un'opportuna interfaccia.

[9] PGN (Portable Game Notation) e EPD (Extended Position Description) sono i due più importanti standard per la memorizzazione di partite di scacchi su file.
[10] Vedi cap. 7.

2 Caratteristiche di un programma di scacchi

2.1 In quali problemi ci si imbatte?

Come già visto nel capitolo uno un programma che gioca a scacchi si scontra con il fattore tempo: il numero di mosse da analizzare nel corso di una partita è talmente grande che se avessimo costruito un programma che gioca la partita perfetta, in grado cioè di vincere sicuramente applicando un algoritmo di teoria dei giochi, e avessimo iniziato la partita nell'istante in cui è nato l'universo, la partita sarebbe ancora in corso. È quindi necessario accontentarsi, cioè rinunciare a valutare la gran parte delle possibili mosse e soprattutto essere consapevoli che sarà la tecnologia di cui disponiamo a limitare la profondità a cui esplorare le mosse possibili.

Un buon programma per scacchi deve necessariamente essere veloce. Questo significa molte cose. Innanzitutto, come abbiamo detto sopra, si dovrà fare in modo di esaminare soltanto una parte dell'albero delle mosse. Oltre a questo, però, bisognerà ottimizzare le strutture dati e le istruzioni, in modo che l'esecuzione di codice non riservato ai calcoli della prossima mossa sia il più rapido possibile.

Un programmatore poco esperto di programmi per scacchi potrebbe, per esempio, in fase di progettazione, stabilire che il suo motore, per rappresentare i pezzi sulla scacchiera, si basi su una struttura dati a matrice, assegnando ad ogni pezzo una costante.

R	**N**	**B**	**Q**	**K**	**B**	**N**	**R**
P	**P**	**P**	**P**	**P**	**P**	**P**	**P**
P	P	P	P	P	P	P	P
R	N	B	Q	K	B	N	R

Qual è il problema di questa rappresentazione? Innanzitutto una costante occupa come minimo 16 bit, questo significa che una matrice del genere occuperà 1 KB di memoria (immaginando di associare le case vuote ad una costante `EMPTY`).

I problemi però peggiorano quando questa tabella va usata, letta, modificata. Se per esempio vogliamo effettuare la mossa *e2e3*, dovremo usare due istruzioni come queste

```
Board[e][2]= EMPTY;
Board[e][3]= PAWN;
```

cancellando la prima volta e scrivendo la seconda informazioni da due byte e dovendo inoltre trattare con due indici. Le cose si complicano rapidamente se consideriamo casi meno banali (ma non per questo poco frequenti), come ad esempio vedere se c'è una torre avversaria sulla prima colonna, oppure stabilire se il re è sotto scacco o ancora voler determinare quali siano le due case occupate dai due cavalli nemici. Operazioni del genere sarebbero molto lente e macchinose (in alcuni casi richiederebbero perfino cicli *while*) e l'obiettivo della snellezza e velocità del programma diventerebbero irraggiungibili.

Gran parte dei programmi per scacchi si basano invece su operazioni logiche bit a bit (AND, OR, XOR e lo SHIFT). Queste operazioni sono infatti le più veloci che un calcolatore può eseguire. Spesso succede che i progettisti scelgano di sprecare più memoria per guadagnare in rapidità di calcolo.

In queste rappresentazioni si usano solo bit, i vettori e le matrici sono composti di elementi che possono assumere come valore soltanto 0 o 1.

Per esempio, per tenere traccia dei pezzi avversari viene memorizzata una matrice[1] con un 1 in corrispondenza del pezzo nemico e 0 altrove.

1	**1**	**1**	**1**	**1**	**1**	**1**	**1**
1	**1**	**1**	**1**	**1**	**1**	**1**	**1**
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Per memorizzare la posizione delle nostre torri si usa una matrice con uno nelle caselle occupate dalle torri e zero altrove.

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	**1**

In questo modo tutte le istruzioni possibili vengono effettuate tramite semplici operazioni logiche bit a bit. Consideriamo

```
bit[2] Board[color][piece]
bit Row[8]
```

dove `Board[color][piece]` è la struttura dati che rappresenta le posizioni dei pezzi di tipo *piece* e di colore *color*, e `Row[8]` è la struttura dati per cui `Row[0]` ha tutti uno sulla prima colonna e zero altrove, `Row[1]` ha tutti uno sulla seconda colonna

[1] Parliamo di matrice solo perché dà l'idea della scacchiera; in realtà i programmi memorizzano array di bit o strutture di bit predefinite dal linguaggio di programmazione usato.
[2] Il tipo *bit* non esiste, almeno nel linguaggio C. Lo usiamo solo per indicare una variabile che può assumere valore zero o uno. Per due tra le possibili dichiarazioni di tale struttura dati vedere i dettagli di GNU Chess e Crafty, capp. 3 e 4.

e zero altrove, e così via. Possiamo determinare se c'è una torre avversaria sulla prima colonna semplicemente con questa istruzione

```
if (Board[black][rook] & Row[1]) ...
```

Se ancora vogliamo muovere il re bianco da *e1* a *e2* è sufficiente l'istruzione

```
Board[white][king] << 8
```

si esegue cioè uno shift di otto posizioni, spostando il bit che rappresenta il re, muovendo idealmente il pezzo dalla prima alla seconda traversa.

Per un motore scacchistico è importante la filosofia secondo cui 'le cose inutili non vanno fatte'. È abbastanza inutile, nonché dispendioso in termini di tempo, rivalutare delle mosse o delle situazioni che si sono già valutate in precedenza. Per questo si usano dei meccanismi per evitare di commettere tali errori, meccanismi che permettono al programma di immagazzinare informazioni sulla partita che potranno tornare utili nel seguito del gioco.

Analizzeremo più approfonditamente tutti questi aspetti nel corso del capitolo.

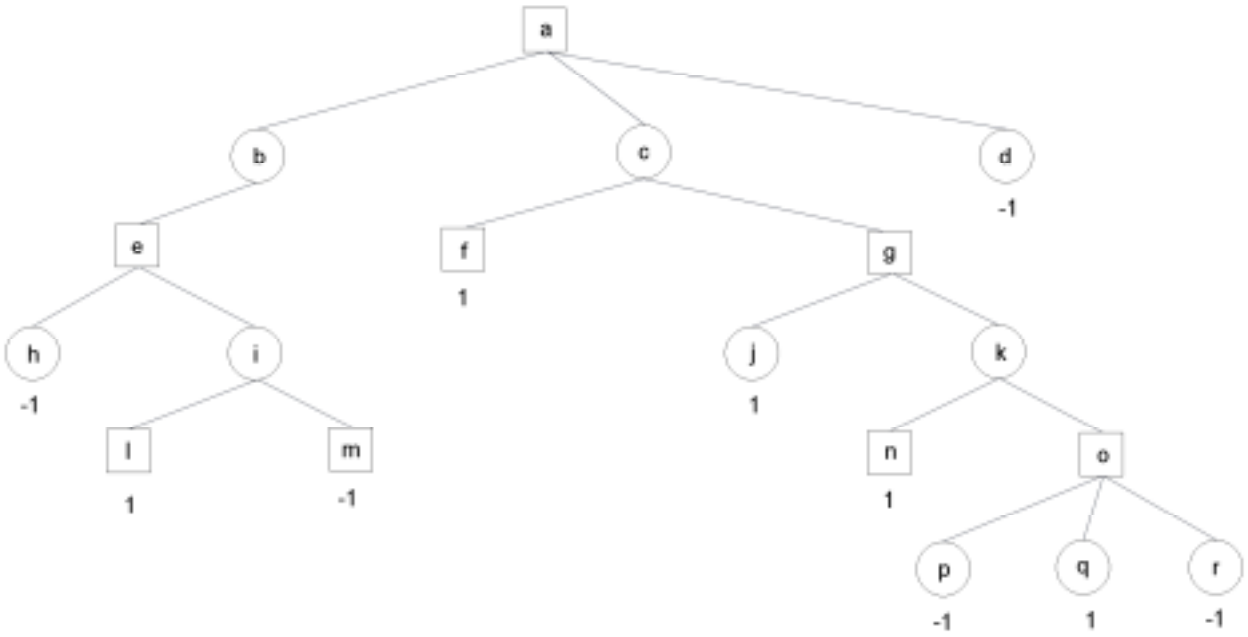
2.2 Gli algoritmi

Gli algoritmi della teoria dei giochi si basano su alberi di mosse. L'idea di base è di partire dallo stato corrente del gioco e generare tutte le mosse successive possibili; poi, per ogni mossa successiva (quindi di profondità uno), si generano tutte le mosse di profondità due, e così via. In questo modo si riesce a generare tutti i cammini possibili che possono svilupparsi da un dato istante in poi ed è quindi possibile scegliere il migliore.

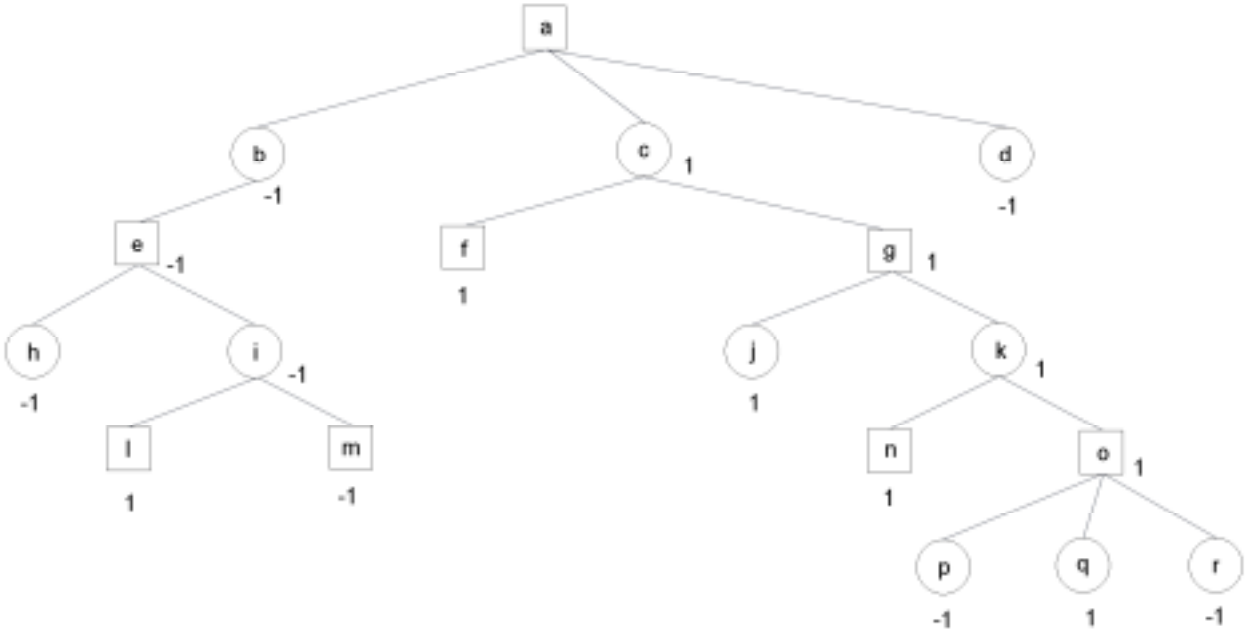
2.2.1 L'algoritmo MinMax

Questo algoritmo effettua una ricerca esaustiva su tutto il sottoalbero delle mosse, fino ai nodi terminali. Una volta raggiunto questo punto si etichettano i nodi favorevoli per il giocatore MAX con 1 e quelli favorevoli al giocatori MIN con -1

(eventualmente si assegna 0 ai nodi pari). Siamo nella situazione in figura (dove il quadrato significa che la mossa è al MAX, il cerchio che la mossa è del MIN):



A questo punto, partendo dai valori associati ai nodi terminali, si risale verso la radice completando le valutazioni per tutti gli altri nodi, secondo la seguente regola: un nodo in cui la mossa è del MAX (i quadrati) ha come label il massimo delle label dei figli, mentre un nodo in cui la mossa è del MIN (i cerchi) ha come label il minimo delle label dei figli. Si arriva così alla seguente situazione:



Quindi alla radice, con tratto al giocatore MAX, viene scelta l'alternativa c, dal momento che porta ad un nodo di valore massimo. Con questo algoritmo siamo in grado di risolvere semplicemente tutti i piccoli giochi a informazione completa (ad esempio il filetto): i nodi terminali con etichetta 1 rappresentano la vittoria di un giocatore, quelli contrassegnati da –1 la vittoria dell'avversario. Ovviamente l'algoritmo funziona anche con valori diversi da –1 e 1 e anche con più di due valori (anzi vedremo in un paragrafo successivo che per gli scacchi siamo 'obbligati' a trattare con valori più eterogenei).

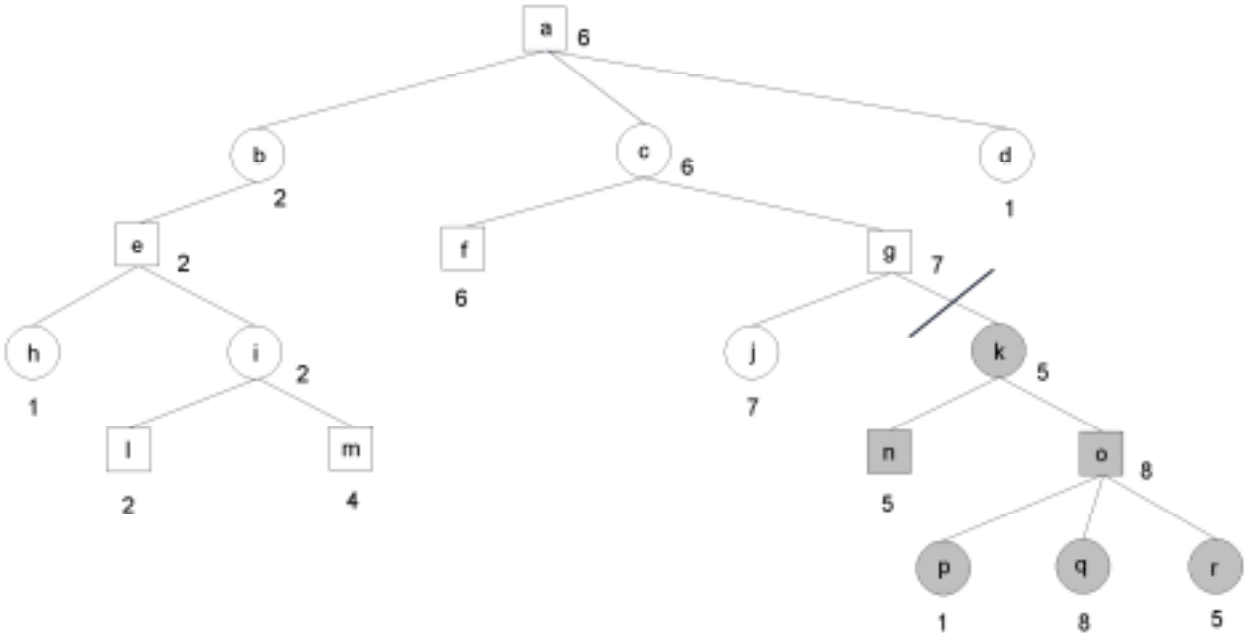
Vediamo l'algoritmo nel dettaglio:

```
int MinMax (node, color)
{
   if (node è terminale)
      return f(node);
   else
   {
      for (ogni figlio son[i] di node)
         val[i] = MinMax(son[i], ~color);
      if (color == white)
         return max(val);
      else
         return min(val);
   }
}
```

L'algoritmo ritorna il valore della funzione di valutazione `f` del nodo dato in input.

2.2.2 Alpha-Beta pruning

L'algoritmo AlphaBeta non è altro che un'ottimizzazione del MinMax. Consiste nel non considerare alcuni cammini dell'albero perché ininfluenti sul risultato. Vediamolo con un esempio:



Supponiamo di voler valutare il nodo c (tratto al MIN). Il figlio di sinistra ha valore 6, ricordiamo che dobbiamo scegliere il valore minimo tra 6 e il valore del figlio di destra. Applicando l'algoritmo MinMax (supponendo di effettuare la valutazione dei nodi terminali sempre da sinistra verso destra) arriviamo a stabilire che il nodo j ha valore 7. Ora, siccome in corrispondenza del padre di j (ovvero g) il tratto è al MAX, MAX sceglierà un valore maggiore o uguale a 7, che è comunque maggiore di 6. Quindi ovviamente MIN sa già, senza continuare la ricerca, che gli conviene scegliere 6, cioè il nodo f. Quindi la valutazione del sottoalbero con radice k è ininfluente e viene evitata.

L'algoritmo prende in input due valori α e β che sono rispettivamente il limite inferiore e superiore del nodo da valutare. Se durante la ricerca troviamo un nodo con mossa al MAX e con valore minore di α allora ci troviamo di fronte ad un nodo ininfluente e operiamo un taglio α, se invece la mossa è al MIN e troviamo un nodo con valore maggiore di β operiamo un taglio β. In mancanza di informazioni i due valori di input di α e β sono $-\infty$ e $+\infty$. Il programma ritorna sempre il valore del nodo in input.

```
int AlphaBeta (node, color, alpha, beta)
{
   if (node è terminale)
      return f(node);
   else
      if (color == white)
```

```
        {
           for (ogni figlio son[i] di node)
           {
              val[i] = AlphaBeta(son[i], ~color, alpha, beta);
              if (val[i] > alpha)
                  alpha = val[i];
              if (val[i] > beta)
                  return INFINITE;
           }
           return alpha;
        }
        else
        {
           for (ogni figlio son[i] di node)
              {
                 val[i] = AlphaBeta(son[i], ~color, alpha, beta);
                 if (val[i] < alpha)
                    return –INFINITE;
                 if (val[i] < beta)
                    beta = val[i];
              }
           return beta;
        }
}
```

2.2.3 L’ordinamento dei nodi

Un modo per migliorare le prestazioni dell’algoritmo AlphaBeta è quello di ordinare i nodi intermedi dell’albero. Se infatti i nodi vengono ordinati in modo tale che le mosse che hanno una valutazione più alta vengano analizzate per prime, aumenta la probabilità che queste mosse possano causare subito un taglio.

Non è necessario ordinare completamente il livello corrente, ma è sufficiente portare in prima posizione la mossa con valore maggiore, nella speranza che le mosse rimanenti non vengano analizzate a causa di un taglio. Se questo non succede, la mossa di valore immediatamente inferiore sarà portata in prima posizione all’iterazione successiva.

L’ordinamento dei nodi intermedi richiede di assegnare loro degli pseudovalori, che saranno naturalmente più rozzi e meno precisi dei valori che scaturiscono dalla funzione di valutazione.

2.2.4 Mosse killer

L’euristica delle mosse killer garantisce a certe mosse che si rivelano particolarmente efficaci in corrispondenza di una certa profondità, di essere valutate prima delle altre.

L’idea è di tenere traccia di queste mosse che causano ripetutamente dei tagli e porle all’inizio della lista di mosse ordinate. Così facendo il programma le valuta per prime, evitando di perdere tempo ad analizzare altre mosse che saranno certamente peggiori.

2.2.5 History Heuristic

Una tecnica simile a quella delle mosse killer è l’History Heuristic: la differenza principale con le mosse killer è che questo secondo metodo non è dipendente dalla ply, cioè dalla profondità dell’albero.

Consiste nel tenere memoria di tutte le mosse migliori che vengono giocate in una tabella 64×64, indicizzata dalla casa di partenza e da quella di arrivo. I valori contenuti nella tabella partono da zero e vengono incrementati ogni qual volta la mossa relativa è giudicata la migliore. Queste informazioni vengono poi utilizzate per ordinare i nodi, secondo il ragionamento che le mosse con un valore più alto nella tabella hanno buona probabilità di essere considerate delle buone mosse anche in futuro.

2.2.6 MVV/LVA (Most Valuable Victim/Least Valuable Attacker)

Un approccio completamente diverso è l’MVV/LVA (Most Valuable Victim/Least Valuable Attacker). È una tecnica statica di ordinamento che pesa le mosse al momento della loro generazione in funzione di eventuali catture. Il valore assegnato è ottenuto come differenza tra il valore del pezzo catturato (eventualmente corretto da un fattore moltiplicativo) ed il valore del pezzo che muove. L’idea è quella di favorire mosse che permettono di catturare i pezzi più pesanti, cercando di muovere i pezzi più leggeri. Nonostante l’evidente approssimazione della valutazione, riconducibile al solo bilancio del materiale, l’implementazione del MVV/LVA è a costo molto ridotto, per cui è molto diffusa in combinazione con altre tecniche.

2.2.7 Iterative deepening

Iterative deepening è una tecnica usata quando il fattore tempo limita la ricerca nell'albero di mosse.

Consiste nel visitare l'albero ad una data profondità n, poi, se c'è ancora tempo, la profondità viene incrementata a n+1 e così via. La ricerca continua fino a che si verifica una delle due condizioni seguenti:

a) si esaurisce il tempo a disposizione,
b) si raggiunge la massima profondità consentita.

2.2.8 L'algoritmo PVS (Principal Variation Search)

Un algoritmo particolarmente efficiente che si basa sull'ordinamento dei nodi dell'albero è il PVS (Principal Variation Search). Consiste nell'utilizzare l'algoritmo AlphaBeta dopo aver ordinato i nodi per aumentare la probabilità di trovare molti tagli, quindi, assumendo che la prima mossa che risulta dall'ordinamento (principal variation) sia davvero la migliore, non c'è nessun bisogno di conoscere il valore degli altri nodi, dobbiamo solo assicurarci che quella mossa sia effettivamente la migliore. Per fare questo, una volta raggiunta la mossa in esame, analizziamo le altre mosse usando l'algoritmo AlphaBeta con una finestra larga uno. Invochiamo cioè l'istruzione

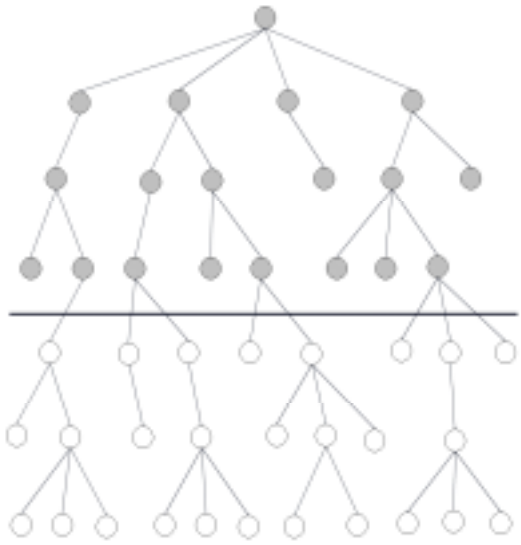
```
val = AlphaBeta(node, color, alpha – 1, alpha)
```

Se la nostra assunzione si è rivelata corretta la ricerca ritorna il valore molto più velocemente che non con una ricerca piena. Se invece l'assunzione era errata e viene trovata una mossa migliore della prima in lista, l'algoritmo procede con la ricerca piena. Questo tuttavia non succede spesso ed è grazie a questo fatto che l'algoritmo si può considerare veramente efficiente.

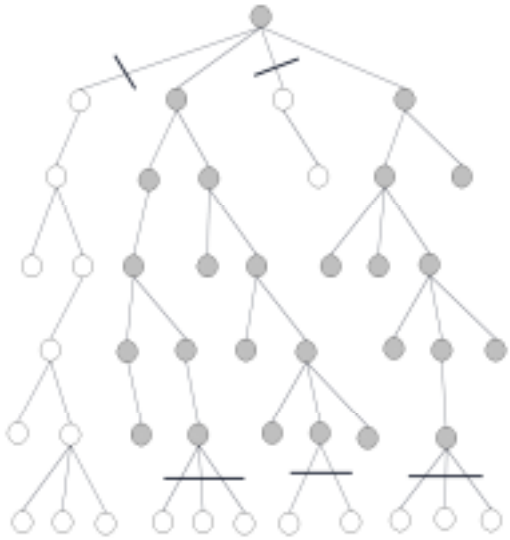
2.2.9 Generazione completa o generazione incrementale?

Nella visita di un albero di mosse ci sono due vie che possiamo intraprendere. La prima strategia prende il nome di generazione completa (conosciuta anche che come espansione cieca o per forza bruta) e consiste nella ricerca esaustiva di tutte le mosse possibili, fino al raggiungimento di una profondità prefissata.

La seconda, detta generazione incrementale (o espansione euristica) prevede di scartare molte delle mosse possibili e per ognuna delle mosse scelte percorrere il relativo sottoalbero fino ad una profondità maggiore del metodo precedente.



generazione completa



generazione incrementale

2.2.10 L’effetto orizzonte

Con l’approccio della generazione completa ci si può imbattere nel cosiddetto effetto orizzonte. Questo deriva da una valutazione sbagliata di una posizione per non aver considerato mosse immediatamente successive a quelle prese in esame. Ciò accade in particolare nel caso di un troncamento di una sequenza di catture. In presenza di una simile eventualità, infatti, la valutazione della posizione è falsata dal temporaneo sbilanciamento del materiale tra le due parti, che permane fino a che tutta la sequenza di catture non è completa. Dopo aver analizzato l’albero alla profondità voluta, quindi, è opportuno continuare la ricerca per i rami corrispondenti alle catture fino a che siano tutte terminate (posizioni quiescenti) o fino ad un secondo livello massimo di ricerca.

2.2.11 Selective extensions

Selective extensions è una tecnica che consiste nello scegliere dei particolari cammini dell'albero e percorrerli in modo più approfondito, a scapito di altri. È un sistema che porta a buoni risultati, ma se realizzato male diventa molto pericoloso, perché rischia di far esplodere le dimensioni dell'albero di ricerca.

Alcuni esempi di extensions possono essere:

- Se il giocatore che ha il tratto è sotto scacco estendi il cammino di una mossa.
- Approfondisci la ricerca quando è stato catturato un pezzo e l'avversario ha effettuato la ricattura del pezzo.
- Se un pedone viene spinto fino alla settima traversa o ha comunque possibilità concrete di promozione estendi il cammino.

2.2.12 L'algoritmo delle mosse nulle

Un'idea relativamente recente[3] e sempre più utilizzata dai programmi per scacchi è l'algoritmo delle mosse nulle. Esso poggia su questa semplice considerazione: supponendo di passare, cioè permettere all'avversario di giocare due mosse consecutive, e vedere che la sua situazione rimane non troppo buona, significa che la situazione iniziale è davvero favorevole per noi. Più formalmente: se il giocatore che ha il tratto decide di passare e la ricerca AlphaBeta per l'avversario ritorna un valore minore di alpha (o, dualmente, la ricerca per il giocatore che ha il tratto è maggiore di beta), viene ritornato semplicemente questo valore: esso rappresenterà un nodo talmente favorevole che possiamo essere quasi sicuri che determinerà un taglio.

Tuttavia si tratta di un metodo rischioso, da usarsi con attenzione, potrebbe infatti dare origine a numerosi errori di natura tattica. È estremamente sconsigliato usare questa tecnica in queste situazioni:

- quando il giocatore che deve muovere ha pochi pezzi rimasti a disposizione,
- quando il giocatore che deve muovere rischia di perdere per scacco matto in un futuro molto prossimo,
- quando il giocatore che ha il tratto è sotto scacco (questo è ovvio, in quanto la tecnica delle mosse nulle genererebbe una posizione illegale).

[3] Donninger 1993.

2.3 Il problema fondamentale

Abbiamo visto che con l'algoritmo MinMax possiamo risolvere qualsiasi gioco a informazione completa. Perché allora non riusciamo a risolvere anche il gioco degli scacchi? Perché, come già spiegato nel capitolo uno, l'albero delle mosse è talmente grande che non riusciamo ad esplorarlo tutto.

Quando abbiamo visto nel dettaglio l'algoritmo MinMax abbiamo detto che i nodi terminali vengono contrassegnati da un 1 se la posizione finale vede la vittoria del giocatore MAX e da –1 se risulta vincente il MIN. Il problema è che negli scacchi non riusciamo ad arrivare fino ai nodi terminali dell'albero, appunto per motivi di complessità, e quindi non riusciamo a etichettare i nodi terminali con 1 o –1 e poi risalire alla radice.

Per ovviare a questo problema siamo obbligati a fermarci prima, a stabilire a priori una profondità che le risorse di cui disponiamo ci consentono di raggiungere e trattare i nodi di quella profondità come fossero le foglie dell'albero. Dopo aver etichettato questi nodi risaliamo verso la radice.

Altro problema: se questi nodi non sono terminali, come facciamo a stabilire chi è il giocatore vincente e etichettare il nodo con 1 o –1? Non possiamo. Dobbiamo in qualche modo assegnare ad ogni posizione un valore sulla base delle informazioni che abbiamo sullo stato del gioco relativamente a quel nodo. Se per esempio raggiungiamo un nodo non terminale che vede il bianco con re e due torri, e il nero con il solo re, non possiamo dire che la vittoria va al bianco, ma possiamo per lo meno asserire che si tratta di una situazione favorevole al bianco e quindi valutare quel nodo di conseguenza.

Da questo segue che ogni nodo verrà etichettato con un valore senza limiti di range, non solo 1 e –1: più tale valore risulta alto più la posizione sulla scacchiera è favorevole per il giocatore in questione.

Come facciamo a dare una valutazione (per forza di cose approssimativa) di un nodo intermedio dell'albero? Con la funzione di valutazione.

2.4 La funzione di valutazione

La funzione di valutazione è una funzione che associa ad ogni possibile posizione sulla scacchiera, relativamente a un giocatore, un valore. Più alto è tale valore più il giocatore si trova in una buona situazione di gioco.

$$f: \text{pos} \rightarrow \Re$$

La funzione di valutazione è uno dei parametri che determinano la bontà di un motore scacchistico. Tale funzione è infatti necessariamente una stima basata su fattori quantitativi, è quindi difficile soppesarli correttamente e riuscire a definire un buon metodo di valutazione.

Vediamo in dettaglio alcuni di questi fattori.

2.4.1 Il materiale

La prima cosa a cui si può pensare per stabilire se una certa posizione sulla scacchiera è favorevole o meno, sono i pezzi nostri e quelli dell'avversario.

È abbastanza ragionevole considerare i pezzi in modo diverso: la regina dà sicuramente più peso di un cavallo, per fare un esempio. Si può pensare quindi di assegnare ad ogni pezzo un valore, per esempio[3]

pedone (p)	1
alfiere (a)	3
cavallo (c)	3
torre (t)	5
regina (r)	9

Dati questi valori il calcolo dello scarto tra il materiale del bianco e quello del nero sarà

$$Mat(pos) = (p_b - p_n) + (a_b - a_n) \times 3 + (c_b - c_n) \times 3 + (t_b - t_n) \times 5 + (r_b - r_n) \times 9$$

[3] Si tratta di valori spesso usati dai programmi per scacchi, almeno in proporzione.

2.4.2 La mobilità dei pezzi

È ovvio che tenere conto unicamente del materiale non è sufficiente. Un altro aspetto rilevante è la mobilità. La mobilità non è nient'altro che il numero di mosse che un giocatore può effettuare. Nel valutare una posizione, quindi, la mobilità è uguale a

$$Mob(pos) = m_b(pos) - m_n(pos)$$

dove m è il numero di mosse possibili.

2.4.3 Posizione del materiale

L'utilità di un pezzo, oltre dal tipo di pezzo e dalle mosse che può effettuare, dipende in maniera sensibile anche dalla sua posizione sulla scacchiera. Tutti sanno per esempio che un cavallo relegato su una colonna di bordo è di poca utilità, mentre lo stesso pezzo posto al centro della scacchiera può effettuare il maggior numero di mosse possibili.

Oltre al posizionamento di pezzi particolari di solito si mira anche ad altri obiettivi, come per esempio il controllo del centro o la zona circostante il re avversario o, per scopi difensivi, la zona in prossimità del re amico.

2.4.4 I pedoni

Molti programmi per scacchi tengono in gran considerazione i pedoni, la loro posizione e le relazioni con i pedoni amici.

Sono sfavorevoli per un giocatore i seguenti casi:

- *pedoni isolati*: sono quei pedoni che non hanno pedoni amici nelle due colonne adiacenti,
- *pedoni doppiati*: si dicono doppiati due pedoni amici sulla stessa colonna,
- *pedoni arretrati*: è arretrato un pedone che nelle due colonne adiacenti non ha un pedone compagno affiancato o più arretrato.

Tali eventualità si traducono in una penalità che va a decrementare il valore fornito dalla funzione di valutazione.

Ci sono anche situazioni favorevoli, come ad esempio i *pedoni passati*, cioè quei pedoni che non hanno pedoni avversari davanti a loro sulla stessa colonna, né sulle due colonne adiacenti, e sono quindi in una posizione favorevole per la promozione.

Ci sono poi altre conformazioni particolari della struttura pedonale che alcuni programmi valutano. Possono essere dati dei bonus per esempio se i pedoni si muovono più o meno assieme in modo coordinato, o se i pedoni attaccano il centro della scacchiera, o ancora se la linea pedonale è disposta in diagonale in modo che tutti siano difesi da un pedone compagno.

2.4.5 Bonus e penalità

Esistono dei bonus (che aumentano il valore di una data posizione) e delle penalità (che lo decrementano) che vengono dati al verificarsi di particolari condizioni.

Per esempio può essere assegnato un bonus se il giocatore ha ancora entrambi i cavalli, o la coppia di alfieri, oppure bonus che vengono assegnati ai pezzi in modo crescente al diminuire della distanza dal re avversario. Un altro esempio possono essere i bonus dati ai pezzi *slider*[4] se tengono sotto attacco un pezzo avversario anche in presenza di ostacoli sul loro cammino (per esempio un pezzo amico del pezzo attaccante), in quanto potenzialmente possono dare origine a situazioni di minaccia all'avversario nel giro di poche mosse.

Un classico esempio di penalità, invece, è dato dalla sopravvenuta impossibilità di arroccare (per esempio perché il re è stato mosso).

Bonus e penalità possono variare dinamicamente col proseguire della gara, oppure possono essere assegnati solo se la partita sta attraversando una particolare fase. Per esempio al re può essere aggiudicato un bonus se nella fase finale della gara si trova in prossimità del centro, mentre può ricevere una penalità per lo stesso motivo ma in fase d'apertura.

[4] Vengono definiti slider quei pezzi che possono effettuare mosse in linea retta o diagonale senza limiti di distanza (alfiere, torre e regina).

2.4.6 Definire una buona funzione

Arrivati a questo punto, mettendo insieme tutti le valutazioni parziali che sono scaturite dal calcolo del materiale, della mobilità eccetera, giungiamo alla definizione di un'unica funzione.

Naturalmente non esiste una soluzione univoca a questo problema, e anche questo compromesso tra informazioni eterogenee può contribuire a rendere un motore per scacchi più efficace.

I valori stabiliti in precedenza possono essere semplicemente sommati tra di loro, ma sembra più sensato bilanciare opportunamente queste valutazioni. Per fare questo è opportuno stabilire dei pesi che vanno a correggere ogni fattore, a seconda dell'importanza che hanno sulla stima della posizione in esame. La funzione di valutazione assume così la forma:

$$f(pos) = Mat(pos) \times p_1 + Mob(pos) \times p_2 + \ldots + TOTbonus - TOTpenalità$$

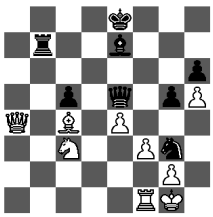
dove $p_1 \ldots p_n$ sono i pesi assegnati ai fattori.

2.5 Tabella delle trasposizioni

Spesso una posizione sulla scacchiera si può ripetere più volte. Si pensi per esempio ai finali di partita, in cui pochi pezzi tornano spesso in posizioni occupate di recente. È senza dubbio uno spreco di tempo analizzare nuovamente una posizione già studiata e ricalcolare la funzione di valutazione. Per garantire che ciò non avvenga quasi tutti i programmi usano una tabella delle trasposizioni. Ogni volta che una posizione è valutata, il valore che ne scaturisce viene memorizzato in questa tabella. Se in futuro tale posizione dovesse ripresentarsi sarà sufficiente ritornare il valore memorizzato nella tabella, con un risparmio considerevole di calcoli.

Naturalmente sarebbe impensabile confrontare direttamente la posizione attuale con tutte quelle precedenti. Per rendere efficiente tale metodo di solito si usano tabelle hash, indicizzate da opportune funzioni hash. La funzione hash deve essere calcolabile velocemente (anche in questo caso, spesso, si usano operazioni logiche bit

a bit) e serve per generare l'indice che permette al programma di accedere alla posizione esatta della tabella in tempo costante, evitando operazioni di confronto.



INDICE	*VALUTAZIONE*
112	3
6546	***8***
430	1
20988	12
…	…

2.6 La rappresentazione della scacchiera

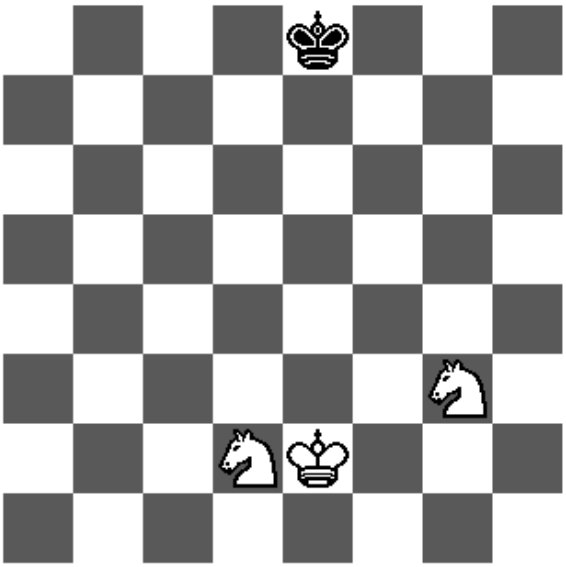
La rappresentazione della scacchiera è una scelta che può influire non poco sull'efficienza del programma. Da essa infatti dipendono la valutazione delle posizioni e la generazione delle mosse.

Se i primi programmi usavano la più intuitiva delle rappresentazioni, cioè una matrice 8×8 in cui ogni elemento rappresentava il contenuto della casa, con il tempo tale approccio è diventato sempre meno frequente a causa di ovvi motivi di efficienza[5].

2.6.1 Bitboard

La tecnica in assoluto più usata dai programmi odierni è la bitboard (o mappa di bit). Come già accennato nella sezione 2.1 una bitboard è un vettore bidimensionale di 64 bit, ognuno associato ad una casa della scacchiera. Con una singola bitboard non si rappresenta tutta la scacchiera ma soltanto la posizione di un determinato tipo di pezzo. Quindi ogni tipo di pezzo avrà la sua bitboard. Vediamo l'esempio della posizione dei cavalli del bianco.

[5] Per qualche esempio vedi sez. 2.1.



0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	**1**	0
0	0	0	**1**	0	0	0	0
0	0	0	0	0	0	0	0

In realtà solitamente una bitboard non è rappresentata da una matrice, piuttosto da un vettore, in cui si definisce un ordinamento tale da associare ogni casa della scacchiera a un elemento del vettore. Per esempio si può adottare la convenzione che le case vengano numerate in questo modo:

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

La casa *a8* in questo caso viene associato al bit più significativo e *h1* al bit meno significativo. In un tale scenario, ad esempio, la bitboard che rappresenta dove si trovano i pezzi bianchi all'inizio della partita è la seguente:

1111111111111111000000000000000000000000000000000000000000000000

Le bitboard non vengono usate soltanto in relazione ai tipi di pezzi e alla loro posizione, ma anche per tenere traccia di moltissime altre informazioni. Per esempio una bitboard può rappresentare tutti i pezzi della scacchiera (solo la posizione,

indipendentemente dal tipo) oppure i pezzi bianchi o quelli neri. Ancora si può associare ad ogni colonna e ad ogni traversa una bitboard con tutti uno sulle caselle della colonna o traversa e zero altrove. In questo modo capire se un pezzo si trova su una determinata colonna, per esempio, diventa un'operazione molto veloce.

Ancora si possono creare opportune maschere che vengono utilizzate per valutare le posizioni sulla scacchiera: per esempio una bitboard con uno sulle case del centro, per la valutazione del controllo del centro.

2.6.2 Esempi

Vediamo qualche esempio per mostrare come si possono derivare informazioni anche complesse effettuando semplici operazioni logiche sulle bitboard.

Esempio 1

Se abbiamo a disposizione le bitboard che rappresentano i vari pezzi del nemico possiamo ottenere una bitboard con tutte le case occupate dall'avversario, indipendentemente dal tipo di pezzo, attraverso un'operazione di *or*.

```
EnemyPieces = b[P] | b[N] | b[B] | b[R] | b[Q] | b[K]
```

Esempio 2

Supponiamo di aver generato le mosse pseudo-legali[6] per i pedoni (`Pawn_Moves`), ottenendo una bitboard con un uno in corrispondenza delle case in cui il pedone può muovere e zero altrove. Consideriamo il caso in cui il pedone muove in avanti di una casa (solitamente i casi di generazione di mosse in avanti o in diagonale sono trattati distintamente). Data la natura del pedone sappiamo che se muove in avanti in linea retta non effettuerà catture. Avendo due bitboard rappresentanti i nostri pezzi e quelli dell'avversario, per assicurarci che la casa di arrivo non sia occupata, avremo bisogno di una istruzione come questa:

```
Moves = Pawn_Moves & ~(MyPieces | EnemyPieces)
```

[6] Vedi sez. 2.7.

Esempio 3

Occupiamoci ora della generazione di mosse di pedoni che avanzano di due case in linea retta. Dobbiamo scoprire quali sono i pedoni che non sono mai stati mossi, per tutti gli altri infatti la mossa è illegale.

```
Moves = ((((MyPawn & (255 << 48)) >> 8) & ~(MyPieces |
EnemyPieces)) >> 8) & ~(MyPieces | EnemyPieces)
```

Vediamo di dare qualche spiegazione. `255 << 48` rappresenta la seconda traversa: infatti il numero 255 in binario su 64 bit è

0000000000000000000000000000000000000000000000000000000011111111

Shiftando a sinistra di 48 bit diventa

0000000011111111000000000000000000000000000000000000000000000000

che, secondo l'ordinamento della bitboard dato nel paragrafo 2.6.1, rappresenta esattamente la seconda traversa, dalla casa *a2* alla casa *h2*. Facendo l'*and* bit a bit tra la bitboard di tutti i nostri pedoni (`MyPawn`) e la seconda traversa otteniamo una bitboard contenente tutti i pedoni che non sono mai stati mossi, quindi quelli per cui è possibile effettuare una mossa di due case in avanti.

A questo punto perché la mossa sia pseudo-legale bisogna che

a) non ci siano pezzi (ne amici né nemici) sulla prima casa successiva a quella dove si trova adesso il pedone, cioè la casa che verrà attraversata, e

b) non ci siano pezzi amici sulla casa di arrivo, altrimenti la mossa sarebbe illegale, e neanche pezzi nemici perché un pedone che muove in avanti non può catturare.

Per soddisfare la condizione *a* prendiamo la bitboard ottenuta precedentemente e la shiftiamo a destra di 8 bit, in questo modo abbiamo spostato i pedoni sulla terza traversa. Ora, come nell'esempio precedente, effettuiamo l'*and* con le case vuote, ossia tutte quelle che non contengono né pezzi amici né pezzi nemici. A questo punto

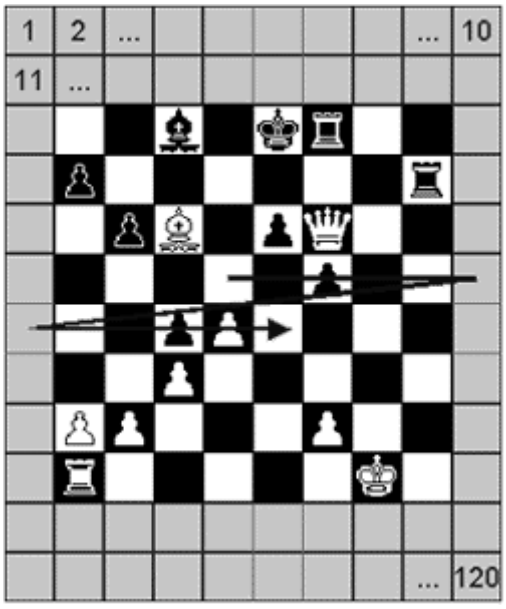
sulla nostra bitboard saranno stati eliminati quei bit rappresentanti pedoni che hanno trovato un ostacolo sulla loro strada.

Per soddisfare la condizione *b* facciamo la stessa cosa: effettuiamo un nuovo shift, giungiamo sulla quarta traversa e rifacciamo il controllo.

2.6.3 Rappresentazione 12×10

Per facilitare il controllo che una mossa pseudo-legale cada all'interno della scacchiera alcuni programmi usano la rappresentazione 12×10. Si tratta di rappresentare la scacchiera con un vettore di 120 elementi, in cui le caselle che si trovano al di fuori della scacchiera vengono etichettati con un valore convenzionale, ad esempio –1. All'atto della generazione delle mosse si effettua il controllo che la casa di arrivo non contenga un –1, altrimenti la mossa viene tolta dalla lista.

Considerando la mossa del cavallo sono necessarie due colonne e due traverse in più su ogni lato. Per la natura sequenziale del vettore, però, le colonne sul bordo destro e sinistro vengono attraversate in successione, quindi è possibile farle collassare e utilizzare solo una colonna in più per lato. Da questo diventa sufficiente una tabella 12×10 anziché 12×12.



2.6.4 Informazioni sullo stato del gioco

È forse superfluo sottolineare che non è sufficiente usare più bitboard per permettere ad un programma di giocare a scacchi. Serviranno infatti molte altre informazioni che non hanno a che fare con la posizione sulla scacchiera. Queste informazioni vengono mantenute in comuni variabili, vettori, alberi eccetera. Facciamo qualche esempio:

- La possibilità per entrambi i giocatori di arroccare.
- La possibilità, con la mossa corrente, di effettuare una cattura en passant.
- Il giocatore che ha il tratto.
- La lista di tutte le mosse finora giocate (per permettere operazioni di *undo*).
- La posizione del re amico e nemico (usata per velocizzare i controlli, evitando di memorizzare un'intera bitboard).

2.7 Generazione di mosse pseudo-legali

Quando l'avversario ha effettuato la mossa il programma si trova a dover decidere con quale mossa controbattere. Per giungere all'albero di mosse e poter così applicare gli algoritmi precedentemente visti, bisogna prima generare le mosse possibili. Di solito si usa il seguente approccio: si generano tutte le mosse possibili che rispettano le caratteristiche dei movimenti di ciascun pezzo, indipendentemente dal fatto che possano risultare legali (per esempio perché muovendo un pezzo si lascia il proprio re sotto scacco). Proprio per questo motivo vengono definite pseudo-legali.

Per fare questo solitamente vengono utilizzate delle mappe che, data una casa della scacchiera e il tipo di pezzo che si vuole muovere, permettono di risalire, in modo molto efficiente dal punto di vista del tempo di elaborazione, a tutte le case raggiungibili dalla posizione di partenza.

Un possibile metodo, molto utilizzato, è quello dei *vettori di movimento*. Per ogni tipo di pezzo si tengono in memoria dei vettori, con un numero di elementi pari alle direzioni che può prendere il pezzo e per ogni direzione un valore che permette di stabilire la casa di arrivo. Facciamo l'esempio del vettore di movimento di un cavallo.

```
V[knight] = (-17, -15, -10, -6, 6, 10, 15, 17)
```

Gli elementi sono 8, infatti, ponendo un cavallo su una casa qualsiasi della scacchiera, le direzioni che può prendere sono 8. I valori rappresentano l'offset tra la casa di partenza e quella di arrivo.

Supponendo un ordinamento della bitboard come quello in sezione 2.6.1, partendo dal valore associato alla casa e sommando i valori del vettore di movimento si ottengono le case di arrivo. Vediamo l'esempio in cui il cavallo è posto sulla casa 36.

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	**19**	20	**21**	22	23
24	25	**26**	27	28	29	**30**	31
32	33	34	35	***36***	37	38	39
40	41	**42**	43	44	45	**46**	47
48	49	50	**51**	52	**53**	54	55
56	57	58	59	60	61	62	63

Ne risultano proprio le otto mosse pseudo-legali del cavallo.

Naturalmente se invece della casa 36 avessimo considerato ad esempio la casa 49, avremmo visto che delle otto mosse possibili solo quattro sarebbero state mosse pseudo-legali, perché le altre quattro sarebbero cadute fuori del limite della scacchiera. Per risolvere questo problema molti programmi usano la rappresentazione 12×10[7]. Con questo sistema si possono riconoscere immediatamente le mosse che non rientrano nella scacchiera ed eliminarle.

2.8 Le aperture

L'apertura è una fase di gioco che è stata studiata molto approfonditamente dagli appassionati di scacchi. Esistono vere e proprie enciclopedie che raccolgono centinaia di aperture diverse, ognuna identificata da un nome.

Per migliorare sia le prestazioni in termini di forza scacchistica, sia l'efficienza in termini di tempo di risposta, spesso i motori per scacchi vengono dotati di un libro di aperture. Si tratta di un file in cui sono memorizzate le possibili mosse iniziali

[7] Vedi sez. 2.6.3.

dell'avversario, ognuna associata a una o più aperture[8]. Quando l'avversario esegue la prima mossa il programma cerca nel libro di aperture se per quella mossa è conveniente rispondere con un'apertura predefinita. Se la trova risponde via via alle mosse dell'avversario completando la sequenza, se invece la ricerca fallisce il programma risponde dopo aver effettuato una normale ricerca nell'albero delle mosse possibili.

Naturalmente se nel libro viene trovata un'apertura e questa viene utilizzata, fino al completamento della sequenza il programma risponde con tempi brevissimi, quasi istantanei all'occhio dell'utente, proprio perché non viene effettuata nessuna generazione di mosse pseudo-legali e nessuna ricerca nell'albero di mosse.

Un libro di aperture può avere diversi formati. Può essere in formato binario, per esempio, oppure in un formato standard leggibile anche dall'utente (per esempio PGN) e successivamente venire compilato in formato binario comprensibile al programma.

2.9 Motori scacchistici

Se si desidera confrontarsi con diversi motori scacchistici non si deve far altro che andare in rete e scaricarsi qualche programma open source. Ce ne sono un'infinità, ognuno con le sue caratteristiche. Ci sono programmi più o meno forti; esistono siti in cui vengono riportati i risultati dei tornei organizzati tra questi programmi, in modo tale da farsi un'idea su quali sono i migliori.

Solitamente i motori più forti sono quelli commerciali, cioè sviluppati per scopo di lucro e che si devono acquistare. Spesso succede che quando un programma gratuito viene continuamente migliorato e arriva a competere con i motori scacchistici più forti, immediatamente viene acquistato da qualche azienda che lo rende un prodotto commerciale.

Naturalmente i programmi commerciali non solo non sono scaricabili dalla rete, ma non è possibile nemmeno accedere ai dettagli della loro realizzazione: algoritmi, strutture dati, strategie eccetera. Noi perciò analizzeremo, nei prossimi capitoli, solo programmi open source. In particolare approfondiremo l'analisi di due programmi:

[8] Se la mossa fa match con un insieme di più aperture, si può ad esempio scegliere, all'interno dell'insieme, l'apertura da utilizzare in modo casuale.

GNU Chess e Crafty, quest'ultimo uno dei più forti motori scacchistici non commerciali.

2.10 Una lista di motori scacchistici

Ecco una lista molto ricca di motori scacchistici, con relativi indirizzi web. Sono tutti freeware e tutti compatibili con l'interfaccia Winboard/Xboard, che analizzeremo in un capitolo successivo.

1. **31337** - *http://paladijn.dhs.org/chess/31337.html*
2. **Abrok** - *http://home.t-online.de/home/roman.korba/*
3. **Alarm** - *http://www.codenet.se/alarm/*
4. **Aldebaran** - *http://members.xoom.it/xaldebaran/*
5. **Amateur** - *http://wbec-ridderkerk.nl/html/EnginesIndex.html*
6. **Amy** - *http://www.winboardengines.de/html/winboard-detailpages.html*
7. **Amyan** - *http://www.geocities.com/zodiamoon/amyan/*
8. **AnMon** - *http://www.winboardengines.de/html/winboard-detailpages.html*
9. **Ant** - *http://www.winboardengines.de/html/winboard-detailpages.html*
10. **Arasan** - *http://www.arasanchess.org/*
11. **Aristarch** - *http://www.zipproth.com/chess/*
12. **Armageddon** - *http://www.armageddon.szach.pl/*
13. **Asterisk** - *http://www.geocities.com/asteriskchess/*
14. **Averno** - *http://www.geocities.com/josec_averno/*
15. **Awesome** - *http://home.vicnet.net.au/~chess/programs.html*
16. **Ax** - *http://www.geocities.com/axchess/*
17. **Baby Chess** - *http://user.cs.tu-berlin.de/~kunegis/bch/*
18. **BACE (BCE)** - *http://www.cse.msu.edu/~bowronch/*
19. **Baron** - *http://home.wish.net/~pijltjes/*
20. **Beaches** - *http://www.codenet.se/alarm/download.asp*
21. **Beowulf** - *http://www.ast.cam.ac.uk/~cmf/chess/beowulf.html*
22. **Bestia** - *http://www.geocities.com/lyapko/winboard.htm*
23. **Betsy** - *http://userfs.cec.wustl.edu/~lwr1/*
24. **BigBook** - *http://www.winboardengines.de/html/bigbookdetail.html*
25. **BigLion** - *http://www.gemuh.de/*
26. **Bionic Impakt** - *http://www.winboardengines.de/html/winboard-detailpages.html*
27. **Blikskottel** - *http://www.oellermann.com/blikskottel/*
28. **BremboCE** - *http://web.tiscali.it/BremboCE/*
29. **Bringer (Der Bringer)** - *http://www.reubold.onlinehome.de/*
30. **Butcher (Rzeznik)** - *http://republika.pl/markol4/*
31. **Chad's Chess** - *http://members.fortunecity.com/clairesbro/chess/*
32. **Chess-Rikus** - *http://www.rikus.com/chess/*
33. **ChessterfieldCL** - *http://home.datacomm.ch/m.luescher/*
34. **Chezzz** - *http://www.winboardengines.de/html/winboard-detailpages.html*
35. **Cilian** - *http://milin.multimania.com/*
36. **ColChess** - *http://www.ast.cam.ac.uk/~cmf/chess/colchess/colchess.html*
37. **Comet** - *http://members.aol.com/utuerke/comet/*
38. **Cpp1** - *http://www.vanheusden.com/cpp1/*
39. **Crafty** - *http://www.tim-mann.org/crafty.html*

40. **Crux** - *http://titanic.nyme.hu/~wyx/crux/*
41. **Damas** - *http://billpoint.tripod.com/damas.htm*
42. **DChess** - *http://www.geocities.com/SiliconValley/Mouse/3246/*
43. **Deep Trouble** - *http://www.winboardengines.de/html/winboard-detailpages.html*
44. **Defeo** - *http://holandany.20m.com/Holand03.htm*
45. **Delfi** - *http://www.sira.it/msb/delfi.htm*
46. **Dragon** - *http://www.winboardengines.de/html/winboard-detailpages.html*
47. **Duke** - *http://www.lordking.bbk.org/*
48. **Embracer** - *http://www.codenet.se/embracer/*
49. **EnginMax** - *http://www.enginmax.de/*
50. **Esc** - *http://escx.cjb.net/*
51. **EXchess** - *http://home.earthlink.net/~econerd/EXchess.html*
52. **Faile** - *http://faile.sourceforge.net/*
53. **Fauce** - *http://wbec-ridderkerk.nl/html/EnginesIndex.html*
54. **Fortress** - *http://freeweb.econophone.ch/fortress/*
55. **Francesca** - *http://www.btinternet.com/~tom_king/*
56. **Frenzee (ChessCraft)** - *http://www.fys.ku.dk/~fischer/frenzee/frenzee.html*
57. **Freyr** - *http://www.geocities.com/SiliconValley/Lab/3716/projects.html*
58. **Gargamella** - *http://digilander.iol.it/gargamellachess/*
59. **Gaviota** - *http://www.msu.edu/~ballicor/gav*
60. **Genesis** - *http://www.cs.biu.ac.il/~davoudo/*
61. **Gerbil** - *http://www.seanet.com/~brucemo/gerbil/gerbil.htm*
62. **Ghost** - *http://www.ghostchess.de/*
63. **Giveaway Wizard** - *http://www.geocities.com/Colosseum/Field/8203/giveaway.zip*
64. **GNU Chess** - *http://www.tim-mann.org/gnuchess.html*
65. **Golem** - *http://www.geocities.com/TimesSquare/Chaos/9481/*
66. **Green Light Chess** - *http://www.7sun.com/chess/*
67. **GreKo** - *http://bearlodge.webservis.ru/chess/greko.html*
68. **Grizzly** - *http://home.snafu.de/~lawi/*
69. **Gromit** - *http://home.t-online.de/home/hobblefrank*
70. **Gullydeckel** - *http://gully.borriss.com/*
71. **Hagrid** - *http://www.friedelprivat.de/*
72. **Holmes** - *http://www.wbholmes.de/*
73. **Horizon** - *http://www.horizonchess.com/*
74. **Inmichess** - *http://www.go.to/inmichess*
75. **Jester** - *http://www.eiganic.com/*
76. **KACE** - *http://raptor.csc.flint.umich.edu/~baird/progs/progs.html*
77. **King of Kings** - *http://www.kingofkingschess.com/kingofkings/*
78. **KnightCap** - *ftp://samba.anu.edu.au/pub/KnightCap/*
79. **KnightDreamer** - *http://www3.tripnet.se/~owemelin/johan/KnightDreamer.html*
80. **Knightx (Techno Chess)** - *http://technochess.free.fr/*
81. **La Dame Blanche** - *http://www.winboardengines.de/ladameblanche/*
82. **LadyGambit** - *http://www.lordking.bbk.org/*
83. **LambChop** - *http://homepages.caverock.net.nz/~peter/chess.htm*
84. **LaMoSca** - *http://it.geocities.com/pietro_valocchi/lamosca/welcome.html*
85. **LarsenVB** - *http://digilander.iol.it/larsenvb/*
86. **Leila** - *http://www.winboardengines.de/html/leiladetail.html*
87. **List** - *http://www.amateurschach.de/schach/download/download.htm*
88. **Little Goliath** - *http://www.borgstaedt.de/*
89. **LordKing** - *http://www.lordking.bbk.org/*
90. **Matheus** - *http://www.winboard.hpg.ig.com.br/wbuk.htm*
91. **MFChess** - *http://home.texoma.net/~garnax/*
92. **Mint** - *http://www-und.ida.liu.se/~chrso085/mint/*
93. **Monarch** - *http://www.stevemaughan.com/chess.htm*
94. **Monik** - *http://www.geocities.com/SiliconValley/Campus/6258/*
95. **Morphy** - *http://www.winboardengines.de/html/winboard-detailpages.html*
96. **Movei** - *http://www.amateurschach.de/schach/download/download.htm*

97. **Mr Chess (Dr Chess)** - *http://www.geocities.com/superchesscraze/drchess*
98. **MSCP** - *http://brick.bitpit.net/~marcelk/*
99. **Muriel (KDLChess)** - *http://www.freston.org/muriel.html*
100. **Mustang** - *http://www.geocities.com/lyapko/winboard.htm*
101. **Nejmet** - *http://nejmet.multimania.com/*
102. **Nero** - *http://www.mit.jyu.fi/~huikari/*
103. **NoonianChess** - *http://home.attbi.com/~noonian/*
104. **OliThink** - *http://btcips73x1.cip.uni-bayreuth.de/~oliver/*
105. **Ozwald** - *http://www.hut.fi/~jostrovs/*
106. **Pentagon** - *http://chessplus.cjb.net/*
107. **Pepito** - *http://www.winboardengines.de/pepito/*
108. **Phalanx** - *http://dusan.freeshell.org/phalanx/*
109. **Pharaon** - *http://www.fzibi.com/pharaon.htm*
110. **Pierre** - *http://pierre.alexboby.com/*
111. **PolarChess** - *http://home.online.no/~malin/sjakk/*
112. **PreChess** - *http://joselauro.tripod.com/prechess-en.html*
113. **Pyotr** - *http://www.digichess.gr/pyotr/*
114. **Quark** - *http://www.quarkchess.de/*
115. **Queen** - *http://home.wxs.nl/~ammeraal/*
116. **Raffaela** - *http://www.linformatica.com/scacchi.html*
117. **RDChess** - *http://groups.msn.com/RudolfPosch/freewareprogramrdchess.msnw?pgmarket=en-us*
118. **Replicant** - *http://www.fortunecity.com/campus/cottingham/359/*
119. **Requiem** - *http://webusers.siba.fi/~ssalmine/requiem.html*
120. **Resp** - *http://home.t-online.de/home/p_rosendahl/*
121. **RivalChess** - *http://www.redhotpawn.com/rival/*
122. **RoboKewlper** - *http://www.codenet.se/alarm/download.asp*
123. **Sachy** - *http://www.winboard.info/forum/topic.asp?TOPIC_ID=11*
124. **SdBC** - *http://www.sdbsoft.com/sdbchess/*
125. **Siboney** - *http://www.amateurschach.de/schach/download/download.htm*
126. **Sjeng** - *http://sjeng.sourceforge.net/*
127. **Skaki** - *http://home.att.net/~glazarou/skaki.htm*
128. **Small Potato** - *http://members.optushome.com.au/alito/smallpotato.html*
129. **SmarThink** - *http://www.aigroup.narod.ru/*
130. **SnailChess** - *http://crasjid.tripod.com/*
131. **Soldat** - *http://wbec-ridderkerk.nl/html/EnginesIndex.html*
132. **SOS** - *http://www.winboardengines.de/html/winboard-detailpages.html*
133. **SSEChess** - *http://www.ssechess.com/*
134. **StAnderson** - *http://surf.to/standersen/*
135. **Storm** - *ftp://ftp.beutlerhvac.com/pub/*
136. **Strategic Deep** - *http://mgkiler.republika.pl/programy.html*
137. **Tamerlane** - *http://escx.cjb.net/*
138. **Tao** - *http://www.winboardengines.de/html/winboard-detailpages.html*
139. **Terra** - *http://www.amateurschach.de/schach/download/download.htm*
140. **The Crazy Bishop** - *http://remi.coulom.free.fr/*
141. **TheLightning** - *http://hubbethebauch.bei.t-online.de/*
142. **Tikov** - *http://www.winboardengines.de/html/tikovchessdetail.html*
143. **T-Rex** - *http://membres.lycos.fr/refigny63/nf3.htm*
144. **Tristram** - *http://www.galahadnet.com/chess/galahad/*
145. **Trynyty** - *http://titanic.nyme.hu/~wyx/crux/download.htm*
146. **TSCP** - *http://home.earthlink.net/~tckjr/*
147. **Ufim** - *http://www.winboardengines.de/html/winboard-detailpages.html*
148. **Wildcat** - *http://www.geocities.com/lyapko/winboard.htm*
149. **Yace** - *http://home1.stofanet.dk/Moq/*
150. **YAWCE** - *http://sand.mobilixnet.dk/*
151. **Zephyr** - *http://www.winboardengines.de/html/winboard-detailpages.html*

3 GNU Chess

3.1 Generalità

GNU Chess è un progetto iniziato nel 1986 da Stuart Cracraft. È stato liberamente distribuito e migliorato da tutta la comunità di appassionati, portandolo ad essere da un programma mediocre a un programma nettamente migliore.

Il prodotto è diventato parte del GNU Project ed è ora un copyright della Free Software Foundation, che permette a chiunque di modificarlo e ridistribuirlo. L'intento della concessione di queste libertà è incoraggiare gli interessati a mettere mano ad un programma di scacchi, capire come funziona (GNU Chess è molto usato per scopi didattici), dare spunti e idee per sviluppare propri motori scacchistici.

Scritto in linguaggio C e ideato inizialmente per il sistema operativo UNIX, GNU Chess è ora compatibile con molte altre piattaforme. Esistono versioni funzionanti per DOS, Windows 3.1, Windows 95/98, Windows NT, Macintosh, Amiga, VMS, Acorn Archimedes e Atari. È inoltre compatibile con l'interfaccia Winboard/Xboard.

La versione di GNU Chess presa in esame in questo capitolo è la 5.04.

3.2 La rappresentazione della scacchiera

GNU Chess usa bitboard[1]. Una bitboard viene definita come un insieme di 64 bit.

```
typedef uint64_t BitBoard;
```

12 bitboard sono sufficienti a rappresentare tutti i pezzi sulla scacchiera.

```
BitBoard b[2][6]
```

Per esempio se vogliamo accedere alla bitboard che rappresenta le posizioni degli alfieri bianchi scriveremo

```
#define white 0
#define bishop 3

b[white][bishop]
```

Diversamente dall'esempio fatto in sezione 2.6.1 GNU Chess si basa sul seguente ordinamento delle bitboard:

56	57	58	59	60	61	62	63
48	49	50	51	52	53	54	55
40	41	42	43	44	45	46	47
32	33	34	35	36	37	38	39
24	25	26	27	28	29	30	31
16	17	18	19	20	21	22	23
8	9	10	11	12	13	14	15
0	1	2	3	4	5	6	7

Naturalmente, come già spiegato nel capitolo due, le bitboard rappresentanti le posizioni sulla scacchiera non sono sufficienti. Occorrono molte altre informazioni. Di queste informazioni, quelle strettamente legate allo stato della scacchiera in un determinato istante, sono raccolte in un'unica struttura chiamata `Board`. Vediamola nel dettaglio.

[1] Vedi sez. 2.6.1.

```
typedef struct
{
   BitBoard b[2][7];
   BitBoard friends[2];
   BitBoard blocker;
   BitBoard blockerr90;
   BitBoard blockerr45;
   BitBoard blockerr315;
   short ep;
   short flag;
   short side;
   short material[2];
   short pmaterial[2];
   short castled[2];
   short king[2];
} Board;
```

- `BitBoard b[2][7]` sono le bitboard che rappresentano i pezzi sulla scacchiera (viste prima).
- `BitBoard friends[2]` sono i pezzi dello stesso colore: `friends[0]` rappresenta lo schema del materiale bianco sulla scacchiera e `friends[1]` lo schema del materiale nero.
- `BitBoard blocker`, `BitBoard blockerr90`, `BitBoard blockerr45` e `BitBoard blockerr315` rappresentano tutti i pezzi (usate per stabilire se un pezzo può bloccare l'avanzata di un pezzo amico). `blockerr90`, `blockerr45` e `blockerr315` derivano dalla bitboard `blocker` ruotata rispettivamente di 90, 45 e 315 gradi.
- `short ep` contiene (eventualmente) la casa su cui si può effettuare una cattura en passant.
- `short flag` contiene informazioni sui tipi di arrocco che è ancora possibile effettuare: arrocco corto e lungo per il bianco, arrocco corto e lungo per il nero.
- `short side` è il giocatore che ha il tratto.
- `short material[2]` è il valore, per lato, del materiale presente sulla scacchiera escluso il re.
- `short pmaterial[2]` è il valore, per lato, del materiale escluso il re e i pedoni.
- `short castled[2]` assume valore uno se il giocatore del colore relativo ha effettuato l'arrocco, zero altrimenti.
- `short king[2]` rappresenta la posizione dei due re (da 0 a 63).

3.3 Generazione di mosse pseudo-legali

La generazione di mosse pseudo-legali è realizzata con la tecnica degli array di movimento.

Per ogni tipo di pezzo viene memorizzato un vettore con il numero di direzioni che quel pezzo può prendere.

```
static const short ndir[8] = {0, 2, 8, 4, 4, 8, 8, 2};
```

L'ordine dei pezzi è *{empty, pawn, knight, bishop, rook, queen, king, bpawn}*. Da notare che i pedoni neri e quelli bianchi vanno trattati a parte, perché sono gli unici pezzi la cui direzione dipende dal colore.

A questo punto si definiscono i veri e propri vettori di movimento, che in questo caso è una matrice di movimento.

```
static const short dir[8][8] =
{
   {0, 0, 0, 0, 0, 0, 0, 0},
   {9, 11, 0, 0, 0, 0, 0, 0},
   {-21, -19, -12, -8, 8, 12, 19, 21},
   {-11, -9, 9, 11, 0, 0, 0, 0},
   {-10, -1, 1, 10, 0, 0, 0, 0},
   {-11, -10, -9, -1, 1, 9, 10, 11},
   {-11, -10, -9, -1, 1, 9, 10, 11},
   {-9, -11, 0, 0, 0, 0, 0, 0}
};
```

dove i due indici rappresentano rispettivamente il tipo di pezzo e la direzione.

Per generare le mosse viene definita una mappa[2] basata sulla rappresentazione 12×10[3].

[2] Il fatto che questa mappa appaia numerata in modo diverso non contrasta con quanto detto sopra circa l'ordinamento delle bitboard.

[3] Vedi sez. 2.6.3.

```
static const short map[120] =
{
   -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
   -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
   -1,  0,  1,  2,  3,  4,  5,  6,  7, -1,
   -1,  8,  9, 10, 11, 12, 13, 14, 15, -1,
   -1, 16, 17, 18, 19, 20, 21, 22, 23, -1,
   -1, 24, 25, 26, 27, 28, 29, 30, 31, -1,
   -1, 32, 33, 34, 35, 36, 37, 38, 39, -1,
   -1, 40, 41, 42, 43, 44, 45, 46, 47, -1,
   -1, 48, 49, 50, 51, 52, 53, 54, 55, -1,
   -1, 56, 57, 58, 59, 60, 61, 62, 63, -1,
   -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
   -1, -1, -1, -1, -1, -1, -1, -1, -1, -1
};
```

A questo punto, per ogni pezzo e per ogni possibile casa di partenza, viene generata la bitboard delle possibili case di arrivo, controllando opportunamente tramite il sistema della rappresentazione 12×10 che la mossa cada all'interno della scacchiera. A tal proposito è definita una matrice di bitboard

```
BitBoard MoveArray[8][64]
```

dove il primo indice rappresenta il tipo di pezzo e il secondo la casa di partenza. Per esempio, `MoveArray[king][44]` ritorna la seguente bitboard:

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	**1**	**1**	**1**	0	0
0	0	0	**1**	***1***	**1**	0	0
0	0	0	**1**	**1**	**1**	0	0
0	0	0	0	0	0	0	0

Una struttura dati molto utile, al fine del generatore di mosse, è la seguente:

```
BitBoard FromToRay[64][64]
```

I due indici rappresentano la casa di partenza e quella di arrivo. `FromToRay[from][to]` ritorna il segmento `from-to` (estremo `from` escluso, estremo

to compreso). Questa struttura dati serve per verificare che in un determinato segmento non ci siano ostacoli e la mossa possa essere portata a termine.

Se per esempio abbiamo deciso di muovere un alfiere da *a5* a *e1* possiamo controllare se ci sono pezzi che ci ostacolano attraverso l'istruzione

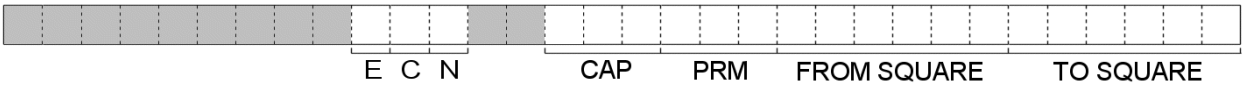
```
If (FromToRay[A5][E1] & Board.blocker) ...
```

Ci sono cinque routine preposte alla generazione di mosse.

- `GenMoves`: routine di base per la generazione di mosse pseudo-legali.
- `GenNonCaptures`: genera solo mosse che non sono catture (le promozioni vengono considerate catture e non vengono generate).
- `GenCaptures`: genera solo catture (incluse catture en passant e promozioni).
- `GenCheckEscapes`: viene utilizzata quando il re è sotto scacco e serve per generare solo quelle mosse che mettono il re in salvo.
- `FilterIllegalMoves`: questa routine si differenzia dalle altre perché si occupa di prendere in input le mosse pseudo-legali, filtrare quelle illegali (per esempio se il re è sotto scacco e lo scacco permane, oppure se muovendo un pezzo si espone il re allo scacco quando prima non lo era) per ottenere alla fine le mosse legali.

Un accenno a come viene memorizzata una mossa da parte del programma.

Una mossa è definita come un *int* a 32 bit.



- *TO SQUARE*: casa di arrivo (da 0 a 63)
- *FROM SQUARE*: casa di partenza (da 0 a 63)
- *PRM*: se è diverso da zero indica che la mossa è una promozione. A seconda del codice presente indica in cosa viene promosso il pedone: cavallo (010), alfiere (011), torre (100) o regina (101).
- *CAP*: se diverso da zero significa che è avvenuta una cattura. Ad ogni possibile pezzo catturato è associato un codice: pedone (001), cavallo (010), alfiere (011), torre (100) e regina (101).
- *N*: mossa nulla.

- *C*: arrocco.
- *E*: cattura en passant.

I bit corrispondenti alle caselle grigie non vengono utilizzati.

3.4 L'albero di mosse

Quando è stata generata una mossa pseudo-legale essa viene memorizzata nell'albero delle mosse. Quando tutte le mosse sono state generate inizia la ricerca per determinare la mossa migliore. I nodi dell'albero sono rappresentati dalla struttura

```
typedef struct
{
   int move;
   int score;
} leaf;
```

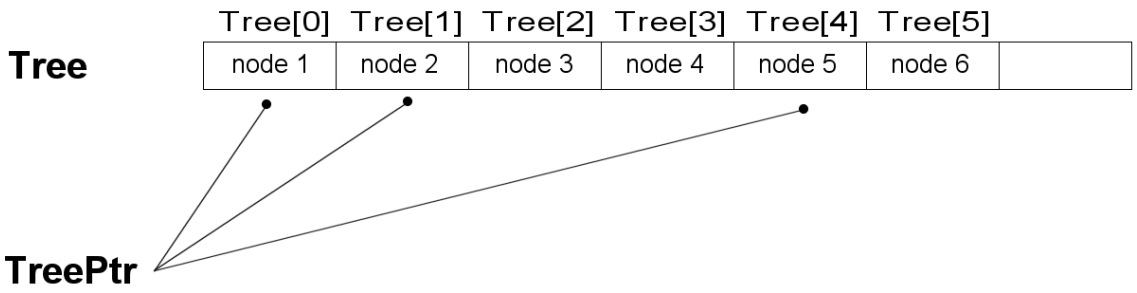
`move` è la mossa, come descritta nel paragrafo precedente, mentre `score` è la valutazione del nodo.

Per l'albero di mosse si usa un vettore e un puntatore, in questo modo:

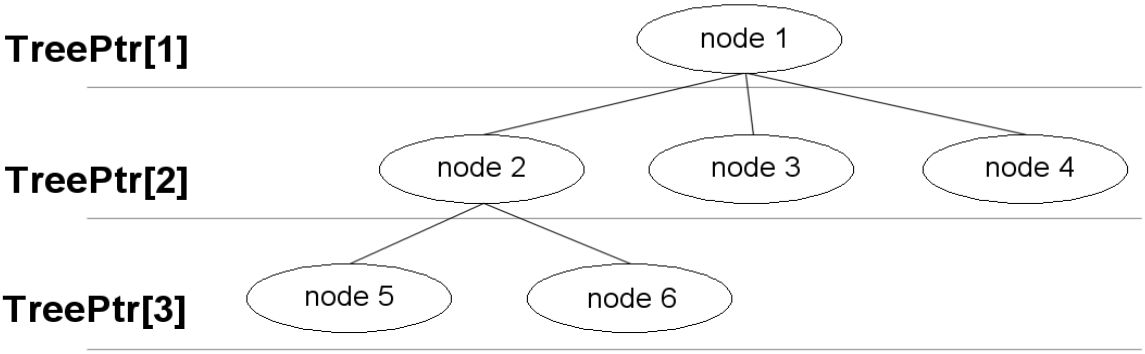
```
leaf Tree[MAXTREEDEPTH];
leaf *TreePtr[MAXPLYDEPTH];
```

dove `MAXTREEDEPTH` è la dimensione massima dell'albero in termini di nodi, mentre `MAXPLYDEPTH` è la profondità massima dell'albero in termini di livelli.

In `Tree` vengono via via memorizzate le mosse, mentre `TreePtr` è il puntatore ai vari livelli di profondità: in questo modo si utilizza il vettore come se fosse un albero.



Ecco come diventa una tale rappresentazione se vogliamo vederla sottoforma di albero.



3.5 Gli algoritmi

GNU Chess usa molti degli algoritmi e delle tecniche viste nella sezione 2.2. Naturalmente in questo paragrafo tali concetti non verranno spiegati nuovamente, ma soltanto menzionati.

GNU Chess utilizza l'algoritmo PVS (Principal Variation Search), effettuando l'ordinamento dei nodi dell'albero secondo varie euristiche (mosse memorizzate nella hash table o nelle tabelle di history, catture, promozioni, euristica delle mosse killer, mosse nulle e mosse verso il centro), migliorando così gli algoritmi MinMax e AlphaBeta. La ricerca continua fino ad una profondità massima, ma è possibile che ci siano delle estensioni di ricerca per alcuni cammini particolari. Le possibili estensioni sono:

- se il re è sotto scacco,
- se il re è minacciato,
- se c'è una cattura seguita da una ricattura sulla stessa casa,
- se ci accorgiamo che l'avversario sta tentando di ritardare lo scacco con mosse inutili.

GNU Chess realizza la tecnica delle mosse nulle, tranne nei seguenti casi:

- quando la mossa precedente era già una mossa nulla,
- a livello uno dell'albero,

- su un nodo candidato ad essere la Principal Variation,
- se il giocatore che muove è sotto scacco,
- se il valore del materiale, compresi i pedoni, (oppure la valutazione della posizione, se questa è memorizzata nella tabella delle trasposizioni) è sotto una certa soglia critica,
- se alla mossa successiva il nostro re finisce sotto scacco,

Infine il programma effettua anche la ricerca quiescente, per evitare che la valutazione venga falsata da una posizione turbolenta. Vengono a questo scopo riconosciute le situazioni critiche, che riguardano scacco al re e catture, e i relativi cammini vengono percorsi per una lunghezza aggiuntiva, fino al raggiungimento di una posizione quiescente.

3.6 Euristica della funzione di valutazione

È interessante vedere nel dettaglio quali sono i fattori che determinano la valutazione di una posizione. Cominciamo dal valore assegnato al materiale.

```
pedone                100
alfiere               350
cavallo               350
torre                 550
regina               1100
re                   2000
```

Il modulo preposto a calcolare la valutazione delle posizioni sulla scacchiera contiene diverse routine. Le elenchiamo tutte commentandole.

- `ScoreP`: valutazione della struttura pedonale. La valutazione è basata sui seguenti fattori:

  _ *pawn square tables,* sono maschere associate ad ogni pedone che servono per verificare se il re avversario è in una zona tale da impedirne la promozione,

_ pedoni passati,
_ pedoni arretrati,
_ pedoni sotto attacco,
_ pedoni doppiati,
_ pedoni isolati,
_ coppie di pedoni passati sulla sesta o settima traversa,
_ pedoni bloccati o pedoni mai mossi,
_ pedoni passati che non possono essere catturati,
_ *pawn storm*, insieme di pedoni che si auto-difendono e attaccano in gruppo.

- `CTL`: valuta il controllo dello spazio, assegnando punteggi diversi a seconda delle zone della scacchiera e dei pezzi che si trovano sulle case relative (per esempio controllo del centro, attacco al re nemico, difesa del re amico, mobilità).

- `ScoreN`: valutazione dei cavalli. Si basa su:

  _ cavalli in prossimità del centro,
  _ cavalli che attaccano il re avversario,
  _ cavalli protetti da pedoni amici,
  _ cavalli che attaccano pedoni avversari deboli.

- `ScoreB`: valutazione degli alfieri. Questi i criteri:

  _ presenza della coppia di alfieri,
  _ mobilità,
  _ controllo dello spazio,
  _ attacco a pezzi avversari,
  _ alfieri protetti da pedoni amici,
  _ posizioni di fianchetto.

- `BishopTrapped`: controlla se vi sono alfieri intrappolati nelle case *a2*, *h2*, *a7* e *h7*.

- `ScoreR`: analisi della situazione delle torri. Vengono effettuati i seguenti controlli:

  _ la presenza di una torre sulla settima traversa e del re avversario sull'ottava, oppure un pedone avversario sulla settima,
  _ torri su colonne o traverse aperte o mezze aperte, cioè prive di ostacoli,

_ controllo dello spazio,

_ torri che tengono sotto attacco pedoni deboli o passati.

- `DoubleQR7`: dà un bonus in presenza di una coppia regina-torre, torre-torre o regina-regina (in seguito a una promozione) sulla settima traversa.

- `ScoreQ`: valutazione della regina. Si basa su questi fattori:

  _ controllo del centro,

  _ distanza dal re avversario,

  _ penalità in mancanza della regina,

  _ attacchi a pedoni esposti.

- `ScoreK`: valutazione del re. Queste le caratteristiche della funzione:

  _ bonus in presenza di una linea di 3 pedoni che difendano il re arroccato,

  _ penalità per la rottura della linea difensiva pedonale prima di un arrocco,

  _ re su una colonna aperta, non difesa da pedoni,

  _ sicurezza del re,

  _ attacco a pedoni avversari,

  _ controllo dello spazio,

  _ re in angolo.

- `LoneKing`: premia il giocatore che ha del materiale diverso dai pedoni mentre l'avversario ha il solo re.

- `KPK`: valuta le posizioni di un finale re-pedone-re.

- `KBNK`: valuta le posizioni di un finale re contro re-cavallo-alfiere.

- `ScoreDev`: analizza la fase di apertura. Valuta i seguenti aspetti:

  _ se il re è arroccato oppure no ed eventualmente se l'arrocco non è più possibile,

  _ sviluppo di cavallo e alfiere,

  _ mosse troppo precoci della regina.

- `Evaluate`: è la funzione centrale, invoca le funzioni precedenti e calcola alcune valutazione preliminari, come il materiale presente sulla scacchiera.

- `EvaluateDraw`: controlla se la posizione corrente è una patta. Per sapere quando dichiarare patta il programma si basa su queste regole:

_ regola delle 50 mosse[4],

_ se sulla scacchiera ci sono dei pedoni non si tratta di una patta,

_ se entrambi i giocatori hanno del materiale di valore inferiore a quello di una torre è patta,

_ se entrambi hanno solo due cavalli o meno è patta,

_ se è un finale re contro re-alfiere-alfiere è patta.

Spesso alcuni bonus o penalità vengono dati solo durante determinate fasi della partita (per esempio in fase di apertura, o durante i finali). Per stabilire che fase sta attraversando la sfida GNU Chess definisce una macro

```
#define PHASE       (8 - (board.material[white] +
                          board.material[black]) / 1150)
```

`PHASE` può quindi assumere valori che vanno da zero (all'inizio della partita) a otto (alla fine), tenendo conto che la divisione di interi viene troncata. Per esempio, l'assegnazione di una penalità in fase d'apertura a causa di arrocco ritardato assumerà la forma

```
if ((PHASE <= 2) && !Board.castled[side])
   score += NOTCLASTLED[5];
```

3.7 Tabelle delle trasposizioni

La tecnica delle tabelle delle trasposizioni viene realizzata con tabelle hash. Ci sono due tabelle hash:

- *HashTab* è la classica tabella delle trasposizioni per memorizzare le posizioni sulla scacchiera,
- *PawnTab* è una tabella hash che controlla le posizioni dei pedoni (la valutazione della struttura pedonale è molto complessa, conviene quindi avere una tabella hash dedicata a questo scopo).

[4] *Regola delle 50 mosse*: se durante una partita vengono giocate 50 mosse consecutive senza che sia avvenuta una cattura o senza che un pedone sia stato mosso la partita viene dichiarata patta.
[5] Si effettua una somma perché la costante `NOTCASTLED` è negativa.

Tali strutture dati sono definite in questo modo:

```
typedef uint32_t KeyType;

typedef struct
{
   KeyType key;
   int move;
   int score;
   short flag;
   short depth;
} HashSlot;
```

```
typedef struct
{
   KeyType pkey;
   BitBoard passed;
   BitBoard weaked;
   short score;
   short phase;
} PawnSlot;
```

```
#define HASHSLOTS 1024
#define PAWNSLOTS 512

HashSlot *HashTab[2];
PawnSlot *PawnTab[2];
```

L'allocazione dello spazio avviene destinando `HASHSLOTS` slots a `HashTab[0]`, `HASHSLOTS` slots a `HashTab[1]`, `PAWNSLOTS` slots a `PawnTab[0]` e `PAWNSLOTS` slots a `PawnTab[1]`.

Ogni `Hashslot` ha una dimensione di 128 bit, il che significa che le hash table (`HashTab[0]` per il bianco e `HashTab[1]` per il nero) occupano ciascuna 16KB, suddivisi in 1024 entrate.

`Pawnslot`, invece, ha una dimensione di 192 bit, quindi le due hash table per i pedoni occupano ognuna 12KB, per 512 entrate.

Un'ultima struttura dati è la seguente:

```
typedef uint64_t HashType;

HashType hashcode[2][7][64];
```

che serve per calcolare la funzione hash, vedremo in seguito come. Questa matrice tri-dimensionale viene inizializzata con dei numeri pseudo-casuali.

Le chiavi per accedere alle due tabelle, generale e pedonale, sono

```
HashType HashKey;
HashType PawnHashKey;
```

Quando ci troviamo a dover valutare una posizione la prima cosa da fare è cercare nella tabella se quella posizione è già stata esaminata in precedenza; in tal caso risparmiamo il tempo di un'elaborazione inutile.

Innanzitutto, data la posizione, dobbiamo calcolare la funzione hash. La funzione hash di GNU Chess è un'operazione di *xor* tra il contenuto di `HashKey` e `hashcode`.

Vediamo un esempio molto rozzo, per chiarire le idee, scritto in pseudo-codice:

```
HashKey = PawnHashKey = (HashType) 0;   /* riempie di 0 */
for (ogni colore)
   for (ogni pezzo)
   {
      considera la bitboard b[colore][pezzo]
      for (ogni casa di b)
      {
         HashKey ^= hashcode[colore][pezzo][casa];
         if (pezzo == pawn)
            PawnHashKey ^= hashcode[colore][pezzo][casa];
      }
   }
```

Alla fine della procedura, vengono messe in *xor* anche informazioni riguardanti il giocatore che muove, le catture en passant e la situazione dell'arrocco. In questo modo tutte le informazioni dello stato della scacchiera vengono compresse in un unico numero da 64 bit.

Ora, quando ci troviamo a dover valutare una posizione, come prima cosa andiamo a vedere se l'informazione che cerchiamo è già presente. Per esempio nella routine di valutazione della struttura pedonale troviamo

```
PawnSlot *ptable;

ptable = PawnTab + PawnHashKey;  /* si posiziona sullo slot */
if (ptable->pkey == PawnHashKey)  /* hash table hit */
   prendi la relativa valutazione;
else                              /* hash table failure */
   calcola funzione di valutazione;
```

Naturalmente in caso di fallimento viene normalmente calcolata la funzione di valutazione e il dato viene aggiornato nella tabella hash con la speranza che sia utile per una valutazione futura.

Nel medio-gioco le prestazioni, in termini di tempo di ricerca, aumentano del 25%-50%, e tale valore cresce se consideriamo i finali. Ancora più sensibile è il

miglioramento dovuto all'utilizzo della tabella dei pedoni: limitandoci a considerare solo quei fattori che dipendono esclusivamente dai pedoni (quindi senza tener conto delle relazioni con gli altri pezzi, per cui la valutazione va fatta ogni volta), la percentuale di hit nella tabella è del 90%-99%.

3.8 Libro di aperture

Dalla versione 5 GNU Chess non viene più distribuito contestualmente al libro di aperture. Si può leggere in un file *readme* che "*il file book.pgn è stato rimosso per motivi di convenienza amministrativa"* e che "*dalla versione 5.x il libro di aperture non verrà incluso nella distribuzione per minimizzare i tempi di download"*.

I libri di aperture si possono comunque scaricare dalla rete o prendere da versioni precedenti dello stesso programma.

Un libro di aperture si può trovare con estensione *.pgn*, per compilarlo è sufficiente usare il comando `"book add book.pgn"` e convertirlo così in formato binario nel file `book.dat`.

Mentre un file in formato PGN è leggibile anche dall'essere umano, un libro di aperture in formato binario è comprensibile alla macchina ed è costituito da record sequenziali con chiavi di tipo `HashType`[6].

Esistono diversi comandi che l'utente può impartire all'interprete per gestire il libro di aperture[7], tra cui la possibilità di attivare o disattivare tale funzione.

3.9 Il formato delle mosse

GNU Chess, se utilizzato senza interfaccia grafica, accetta due formati diversi di mosse,

- notazione CAN (Coordinate Algebraic Notation): per esempio *e2e4*,

[6] Vedi paragrafo precedente.
[7] Vedi sez. 3.11.

• notazione SAN (Standard Algebraic Notation): per esempio *Nf6.*

Quando l'utente inserisce una mossa in notazione algebrica estesa (CAN) questa viene convertita in notazione SAN attraverso un'apposita routine. Se l'utente inserisce una mossa già in notazione SAN la mossa viene esaminata per accertarsi che non vi siano ambiguità (per esempio, se un pedone e un alfiere possono entrambi raggiungere con una mossa la casa *e5* e l'utente inserisce soltanto *e5*, rispettando la notazione, ma senza disambiguarla indicando il pezzo che muove, la mossa non viene accettata).

3.10 Features

Vediamo altre features di GNU Chess che non sono state evidenziate nei precedenti paragrafi, ma che possono essere ugualmente interessanti.

_ UNIVERSAL BOARD

GNU Chess ha la possibilità di interfacciarsi con la Novag Universal Board[8]: si tratta di una scacchiera reale con la quale si può giocare contro il programma muovendo i pezzi con le mani.

_ WINBOARD, XBOARD E INTERNET CHESS SERVER

Il programma è perfettamente compatibile con le interfacce grafiche Winboard/Xboard[9] e può giocare su un Internet Chess Server[10].

_ FILE DI LOG

C'è la possibilità di memorizzare partite, sia dal punto di vista del computer (memorizzando cioè tutti gli output, compreso il cosiddetto *thinking*) in un file `log.nnn`, sia in una notazione più simile a quella PGN in un file `game.nnn`.

[8] Vedi *www.novag.com*
[9] Vedi cap. 6.
[10] Vedi cap. 7.

_ FILE EPD E PGN

Esistono due moduli preposti alla gestione (salvataggio e caricamento) di file contenenti partite formattate secondo i due standard EPD e PGN.

_ DATABASE DI GIOCATORI

È anche possibile gestire liste di giocatori attraverso un semplice database. Per ogni giocatore vengono memorizzati nome, numero di vittorie, numero di sconfitte e numero di patte. Sono possibili le seguenti operazioni sul database: inserimento, cancellazione, modifica, lettura, ricerca e ordinamento.

_ FUNZIONI DI TEST

Servono a) per testare la velocità delle varie funzioni del programma, b) per leggere delle mosse da un file e sottoporle a test di legalità e c) per leggere una posizione da un file e applicare la funzione di valutazione. (Per la lista di comandi relativi alle funzioni di testing vedi il paragrafo successivo.)

3.11 Alcuni comandi

Vediamo un sottoinsieme dei comandi che è possibile impartire all'interprete di GNU Chess.

- `^C` manda un interrupt, interrompe l'azione in corso ed esce dal programma,
- `quit` o `exit` esce dal programma,
- `help` stampa la lista di comandi,
- `book add` compila `book.pgn` in `book.dat`,
- `book on (off)` attiva (disattiva) l'uso del libro di aperture,
- `pgnsave` *`filename`* salva la partita in formato PGN nel file *`filename`*,
- `pgnload` *`filename`* carica il file PGN *`filename`*,
- `force` o `manual` fa sì che il programma non produca più mosse; in questo modo l'utente può continuare a inserire mosse per raggiungere una posizione desiderata.
- `white (black)` il programma gioca col bianco (nero),
- `post` ad ogni mossa il programma produce tutto l'output (risultati della ricerca, varianti, profondità, valutazione delle posizioni, tempo…),

- `nopost` disattiva la funzione precedente,

- `name` *`name`* permette di inserire il nome; successivamente salva la partita nei file `log.nnn` e `game.nnn`,

- `new` resetta lo stato della scacchiera per una nuova partita,

- `hash on (off)` abilita (disabilita) l'uso di tabelle di trasposizione,

- `hashsize` *`n`* aggiorna il numero di entrate della hash table a *`n`*,

- `null on (off)` attiva (disattiva) l'euristica delle mosse nulle,

- `xboard on (off)` abilita (disabilita) l'uso dell'interfaccia grafica xboard,

- `depth` *`n`* impone che la ricerca della mossa ottima abbia una profondità massima di *`n`* ply,

- `level` *`moves minutes increment`* setta i vincoli di tempo a *`moves`* mosse in *`minutes`* minuti, con un incremento di *`increment`* per ciascuna mossa,

- `load` o `epdload` carica un file in formato EPD da disco,

- `save` o `epdsave` salva un file in formato EPD su disco,

- `switch` inverte i colori dei pezzi,

- `undo (remove)` annulla l'ultima mossa (le ultime due mosse),

- `show board (time) (eval)` visualizza la scacchiera (il tempo) (la valutazione della posizione),

- `test movelist` legge da un file EPD indicando quali sono le mosse legali,

- `test capture` legge da un file EPD indicando quali sono le catture legali,

- `test movegenspeed` testa la velocità del generatore di mosse,

- `test capturespeed` testa la velocità del generatore di catture,

- `test eval` legge una posizione da un file EPD e ne ritorna la valutazione,

- `test evalspeed` testa la velocità della valutazione delle posizioni.

4 Crafty

4.1 Generalità

Crafty è un motore scacchistico creato da Bob Hyatt. È diretto discendente di *Cray Blitz*, storico programma che fu anche campione del mondo nel 1983, nel 1986 e nel 1989, creato dallo stesso Hyatt.

Si tratta di un programma molto forte, il più forte motore scacchistico freeware, l'unico in grado di competere con i programmi commerciali. È quindi molto più forte di GNU Chess, visto nel capitolo precedente.

Crafty è scritto in linguaggio C ed è molto portabile; ecco una lista (presa dalla documentazione del prodotto) delle piattaforme su cui è in grado di girare:

- DEC alpha running OSF/1-Digital Unix
- any Cray-1 compatible architecture including XMP, YMP, C90, etc.
- HP workstation running HP_UX operating system (unix)
- PC running DOS, Windows, OS/2 Warp, using DJGPP port of gcc to compile
- RS/6000 running AIX (unix)
- Sun SparcStation running Solaris (SYSV/R4) Unix
- Sun SparcStation running SunOX (BSD) Unix
- Any architecture running the Linux operating system
- Microsoft Win95/WinNT, when compiled with Microsoft Visual C++
- Macintosh and other MacOS-compatible computers

Crafty è compatibile con l'interfaccia Winboard/Xboard, che vedremo in un capitolo successivo.

In questo capitolo considereremo la versione 18.9 di Crafty.

4.2 Rappresentazione della scacchiera

Anche Crafty usa bitboard per rappresentare la scacchiera.

```
typedef unsigned long BITBOARD
```

L’ordinamento della bitboard è uguale a quello usato da GNU Chess: il bit meno significativo è associato alla casa *a1*, fino al bit più significativo associato alla casa *h8*. Una posizione sulla scacchiera è identificata attraverso una struttura `POSITION`.

```
typedef  struct {
   BITBOARD       w_occupied;
   BITBOARD       b_occupied;
   BITBOARD       occupied_rl90;
   BITBOARD       occupied_rl45;
   BITBOARD       occupied_rr45;
   BITBOARD       rooks_queens;
   BITBOARD       bishops_queens;
   BITBOARD       w_pawn;
   BITBOARD       w_knight;
   BITBOARD       w_bishop;
   BITBOARD       w_rook;
   BITBOARD       w_queen;
   BITBOARD       b_pawn;
   BITBOARD       b_knight;
   BITBOARD       b_bishop;
   BITBOARD       b_rook;
   BITBOARD       b_queen;
   BITBOARD       hash_key;
   unsigned int   pawn_hash_key;
   int            material_evaluation;
   signed char    white_king;
   signed char    black_king;
   signed char    board[64];
   signed char    white_pieces;
   signed char    white_minors;
   signed char    white_majors;
   signed char    white_pawns;
   signed char    black_pieces;
   signed char    black_minors;
   signed char    black_majors;
   signed char    black_pawns;
   signed char    total_pieces;
} POSITION;
```

Sostanzialmente non c’è molta differenza con GNU Chess, soltanto il fatto che non vengono usati vettori di bitboard (eccetto `signed char board[64]` dove a ogni casa è associata una bitboard inizializzata con il pezzo presente su essa), ma ogni informazione ha una bitboard dichiarata a parte con un nome significativo. Inoltre possiamo vedere come ogni posizione abbia la propria chiave hash direttamente all’interno della struttura che la rappresenta.

4.3 Generazione di mosse pseudo-legali

Anche Crafty implementa la tecnica dei vettori di movimento. Per mezzo di essi vengono riempite delle tabelle chiamate *attack board*, una per ogni tipo di pezzo, che rappresentano le mosse pseudo-legali.

A differenza di GNU Chess, Crafty non usa la rappresentazione 12×10 per controllare che una mossa non finisca fuori dall'area della scacchiera. In modo molto più intuitivo invece, all'interno del ciclo che inizializza le attack board, calcola la casa di arrivo `sq` a partire dai vettori di movimento ed esegue il seguente controllo

```
if ((sq < 0) || (sq > 63)) continue;
```

evitando eventualmente di calcolare la mossa se la casa di arrivo non è tra le 64 della scacchiera.

Per gli attacchi dei pezzi slider Crafty utilizza lo stesso metodo di GNU Chess: una tabella `obstructed[from_square][to_square]` contiene i vettori di bit che un pezzo attraversa per attaccare la casa `to_square`; per capire se la casa `to_square` è effettivamente attaccabile o raggiungibile è sufficiente mettere tale vettore in *or* con la bitboard delle case occupate.

Il generatore di mosse è formato da 3 routine:

- `GenerateCaptures`: genera catture e promozioni di pedoni.
- `GenerateCheckEvasions`: genera mosse in caso di scacco al re. Tre sono i tipi di mosse considerate in questo caso:

  _ il re viene spostato in una casa sicura (può catturare oppure no),

  _ un pezzo viene interposto tra il re e il pezzo attaccante (a condizione che il pezzo nemico sia uno slider e che sia l'unico che tiene il re sotto scacco),

  _ si tenta di catturare il pezzo attaccante (se i pezzi che tengono sotto scacco il re sono più di uno si tenta di catturarne uno con il re, a patto che la casa di arrivo sia sicura).

- `GenerateNonCaptures`: genera solo mosse che non siano catture.

Esiste poi una funzione `ValidMove` che effettua ulteriori controlli sulla validità statica della mossa (ma non sulla legalità), come per esempio il colore del pezzo che muove, correttezza dell'arrocco, correttezza di catture en passanti, controlli sulla casa di partenza e su quella di arrivo.

Il formato delle mosse è analogo a quello di GNU Chess: una mossa è rappresentata da un intero di 32 bit, dove nei 6 bit meno significativi viene memorizzata la casa di partenza, nei 6 successivi la casa di arrivo, nei bit numero 12, 13 e 14 il pezzo che muove e nei bit restanti vengono alzati dei flag in base alle caratteristiche della mossa (cattura, promozione, arrocco eccetera).

4.4 L'albero di mosse

Esiste una struttura chiamata `TREE`, molto complessa, che tiene memoria di tutte le informazioni sullo stato di gioco in un determinato istante, compreso l'albero di mosse durante una ricerca.

L'albero viene memorizzato in una variabile

```
int *last[MAXPLY];
```

dove ogni elemento è un puntatore a una mossa, cioè un vettore di mosse. Quando le mosse pseudo-legali vengono generate, la routine di generazione relativa memorizza tutte le mosse in un vettore di tipo `int*`, che viene ritornato alla fine della procedura. Questo vettore viene aggiunto a `last[]` e rappresenta un livello (ply) dell'albero di mosse.

4.5 Gli algoritmi

Crafty realizza, per la ricerca della mossa migliore, l'algoritmo PVS (Principal Variation Search). L'euristica usata per l'ordinamento delle mosse si basa su catture, materiale, metodo delle mosse killer, metodo delle mosse nulle e History Heuristic.

L'albero viene visitato secondo la tecnica dell'iterative deepening, estendendo cioè ogni volta l'orizzonte di ricerca, fino a che il tempo non termina. L'ultimo livello visitato viene quindi analizzato e se vengono trovati nodi turbolenti i relativi cammini

vengono percorsi per qualche mossa ancora, fino a raggiungere una posizione quiescente. Analogamente vengono realizzate extensions di alcuni cammini nel caso si possano verificare determinate condizioni particolari.

La routine `Search` realizza la ricerca AlphaBeta con il metodo iterative deepening, controllando ogni tanto la situazione del tempo.

La funzione controlla innanzitutto per ogni ply se il relativo punteggio è presente o meno nella tabella delle trasposizioni, così da risparmiare tempo nella ricerca. Se nella tabella delle trasposizioni non è stata trovata alcuna valutazione si tenta di trovare immediatamente un taglio nell'albero con la tecnica delle mosse nulle. Le mosse nulle non vengono generate nei seguenti casi:

- il giocatore che muove è sotto scacco,
- la mossa precedente era già una mossa nulla,
- il giocatore che muove ha poco materiale sulla scacchiera.

A questo punto la funzione chiama se stessa ricorsivamente, effettuando piccole ricerche ad una profondità sempre più bassa. Questo accade a patto che il tempo non scarseggi, altrimenti viene invocata la routine di ricerca di posizioni quiescenti.

Il programma quindi valuta se ci sono possibili extensions da seguire; le possibilità sono:

- re sotto scacco,
- cattura seguita da ricattura,
- pedoni destinati alla promozione.

La routine `Quiesce` viene invocata dalla funzione `Search` quando la ricerca in profondità viene interrotta.

La routine percorre soltanto quei cammini che partono da nodi turbolenti, ossia nodi per i quali una sequenza di catture non è ancora terminata, sbilanciando così il materiale.

Vengono quindi presi in considerazione soltanto quei particolari nodi e per ciascuno vengono generate tutte le possibili catture, esplorando il cammino di un'unità. Dopo di che la routine invoca se stessa ricorsivamente generando sempre catture, fino al raggiungimento di una posizione quiescente.

Una caratteristica di Crafty non presente in GNU Chess è il cosiddetto *pondering*, ovvero la possibilità, per il programma, di effettuare calcoli anche mentre l'avversario sta pensando a quale mossa fare.

Il programma fa una sorta di predizione su quale potrà essere la mossa dell'avversario. Per fare questo innanzitutto controlla nella tabella delle trasposizioni se, associata a tale posizione, c'è una mossa consigliata; se non c'è esegue una piccola ricerca per trovare una mossa buona che abbia una discreta probabilità di essere quella che l'avversario si sta apprestando a giocare. Una volta scelta la mossa più probabile inizia la normale ricerca per trovare la mossa con cui rispondere.

Mentre il programma sta effettuando il pondering possono avvenire tre cose:

- l'avversario fa la sua mossa, che si rivela essere proprio quella che aveva predetto la routine: in questo caso si esegue uno switch da *pondering* a *thinking*, continuando la normale ricerca nell'albero. Il programma in questo modo ha guadagnato tempo;

- l'avversario muove in maniera diversa da quello che la routine aveva previsto: il pondering si interrompe con un fallimento e inizia la normale ricerca;

- viene inserito un comando: se si tratta di un comando semplice il pondering può continuare e il comando venire soddisfatto; se è un comando che non consente al programma di fare altro il pondering viene interrotto ed eventualmente ripreso una volta soddisfatto il comando, se questo è ancora possibile.

4.6 Symmetric MultiProcessing

Crafty supporta il calcolo parallelo; tale funzionalità è detta Symmetric MultiProcessing (SMP). Chi ha a disposizione una macchina a più processori può compilare il sorgente del programma con i flag `–DSMP` e `–DCPUS=N`, dove `N` è il numero di CPU a disposizione.

Nella funzione `Iterate`, la quale inizia la ricerca nell'albero di mosse invocando la funzione `Search`, vista precedentemente, se la variabile `SMP` è settata, e quindi se ci

si trova su una macchina che supporta il calcolo parallelo, la ricerca viene affidata a dei thread che lavorano in parallelo. A tal fine vengono chiamate tante *fork* quanti sono i thread che abbiamo a disposizione.

Quando il programma si accorge che uno o più thread è in stato *idle*, attraverso la funzione `Search` il thread viene riattivato e lo stato della ricerca dell'albero viene copiato affinché sia a disposizione del nuovo thread, che va ad aggiungersi agli altri per continuare la ricerca in parallelo.

Non appena si è sicuri di aver trovato un taglio e la ricerca va terminata, viene invocata una funzione che sospende tutti i thread dal loro compito e li mette in uno stato di *waiting*, in attesa che inizi la ricerca successiva.

I due moduli `search` e `searchmp` sono sostanzialmente identici (vedi il paragrafo precedente per una descrizione della funzione `Search`); il primo viene utilizzato in caso di ricerca normale, il secondo in caso si stia utilizzando il calcolo parallelo.

4.7 Euristica della funzione di valutazione

Vediamo innanzitutto il valore che *Crafty* assegna al materiale.

```
pedone                100
alfiere               300
cavallo               300
torre                 500
regina                900
re                  40000
```

Analizziamo ora nel dettaglio le routine del modulo preposto alla valutazione della scacchiera.

- `EvaluateDevelopment`: valuta la bontà dello sviluppo dei pezzi da parte del programma nelle prime fasi di gioco. Il programma viene incoraggiato a:

  _ non muovere la regina prima che i pezzi minori siano stati sviluppati,
  _ sviluppare i pedoni centrali,
  _ non muovere il re se non per l'arrocco.

• `EvaluateMate`: viene usata nei finali, per valutare quelle posizioni in cui un giocatore ha il solo re mentre l'altro ha materiale sufficiente per raggiungere lo scacco matto.

• `EvaluateMaterial`: calcola il valore del materiale presente sulla scacchiera.

• `EvaluateKingSafety`: valuta la condizione del re. Si basa sui seguenti fattori:

_ i pezzi difensori,

_ se il re è arroccato valuta la situazione dei pedoni che ha di fronte. Se non lo è valuta comunque i pedoni di lato e cerca di preservarli per un futuro arrocco,

• `EvaluatePassedPawns`: valuta la situazione dei pedoni passati;

_ dà un bonus a quei pedoni che hanno raggiunto la sesta traversa e che sono destinati alla promozione,

_ dà un bonus alla coppia di pedoni passati, uno sulla sesta e uno sulla settima traversa.

• `EvaluatePassedPawnRaces`: è usata per valutare quelle situazioni in cui un giocatore ha un pedone in corsa per la promozione, mentre l'avversario cerca di catturarlo tentando di bloccare il pedone prima che arrivi sull'ottava traversa; la routine tratta anche il caso in cui entrambi i giocatori hanno uno o più pedoni destinati alla promozione.

• `EvaluatePawns`: calcola il punteggio della struttura pedonale. Questi i passi principali della funzione:

_ controlla se la valutazione è già presente nella tabella hash per i pedoni,

_ se l'avversario ha ancora tutti gli otto pedoni il programma viene penalizzato per non essere riuscito ad aprire una colonna,

_ valuta lo sviluppo dei pedoni: i pedoni centrali vengono incoraggiati ad avanzare, mentre quelli laterali a difendere un eventuale arrocco,

_ valutazione dei pedoni isolati,

_ valutazione dei pedoni bloccati,

_ valutazione dei pedoni deboli (un pedone è considerato debole se avanzando di una o due traverse non è più difeso da un pedone dello stesso

colore ed eventualmente è difeso da altri pezzi un numero minore di volte rispetto a quanto è attaccato),

_ analisi dei pedoni doppiati,

_ esame delle colonne aperte o mezze aperte, che diminuiscono il grado di sicurezza del re.

- `Evaluate`: è la funzione principale che invoca le precedenti, oltre ad eseguire le seguenti valutazioni:

_ alfieri intrappolati,

_ torri intrappolate,

_ distanza dei pezzi dal re avversario,

_ valutazione degli alfieri

- mobilità,
- coppia di alfieri,
- controllo del centro,
- posizioni di fianchetto di fronte al re arroccato,

_ valutazione delle torri

- torre su una colonna aperta,
- mobilità,
- torre che difende un pedone passato,
- coppia di torri o torre-regina sulla settima traversa e pedone avversario sulla settima traversa e re avversario sulla settima o ottava traversa,

_ valutazione della regina,

- controllo del centro,
- posizione nei confronti del re nemico,
- regina sulla settima traversa supportata da una torre.

Esterna al modulo di valutazione delle posizioni esiste anche una funzione `Swap` che valuta quando una cattura o una sequenza di catture è favorevole.

Vediamo come il programma valuta se una data posizione viene considerata una patta:

- se entrambi i giocatori hanno dei pedoni non è una patta,

• se nessun giocatore ha dei pedoni si controlla il valore del materiale rimanente: uno dei due deve avere un vantaggio sufficiente per garantirsi la vittoria, altrimenti è patta,

• se un giocatore ha almeno un alfiere e un cavallo non si tratta di patta,

• se un giocatore ha una torre e l'avversario ha un pezzo minore e dei pedoni, la torre non ha possibilità di vittoria e viene decretata la patta,

La funzione `RepetitionCheck` ha il compito di decretare la patta per uno di questi motivi:

- regola delle 50 mosse,
- per ripetizione di mosse[1].

Diamo uno sguardo a quali criteri segue la routine `ResignOrDraw` per decidere di terminare una partita per abbandono o per offrire la patta all'avversario.

Queste decisioni vengono prese sulla base dei risultati della valutazione della posizione e sul tempo rimanente all'avversario. Se la ricerca ha dato come esito che il giocatore artificiale si trova in una situazione molto sfavorevole, cioè se il punteggio è inferiore a una certa soglia, viene decretata la vittoria dell'avversario per abbandono. Se invece il valore ritornato è esattamente uguale al valore che decreterebbe una patta, il programma offre la patta se all'avversario rimangono più di 30 secondi di tempo.

Infine osserviamo come Crafty determina in quale fase di gioco si trovi una partita. Mentre GNU Chess fa uso di una macro che cambia dinamicamente perché si basa sul valore del materiale presente sulla scacchiera in un determinato istante, Crafty utilizza una routine `Phase` che valuta anche aspetti diversi dal valore del materiale, almeno per quanto riguarda la fase di apertura.

Se alcuni pezzi minori sono ancora sulla loro casa di partenza e se entrambi i re non hanno ancora effettuato l'arrocco, pur avendone ancora il diritto, la partita si trova in fase d'apertura.

Se il computo totale del materiale scende al di sotto del valore soglia di 1700 (per esempio due torri e due pezzi minori) la partita entra nella fase finale.

Se la partita non è né in fase d'apertura né in fase finale significa che è in fase di medio-gioco.

[1] Una partita viene decretata patta se una certa posizione si ripete sulla scacchiera per tre volte, anche non consecutive.

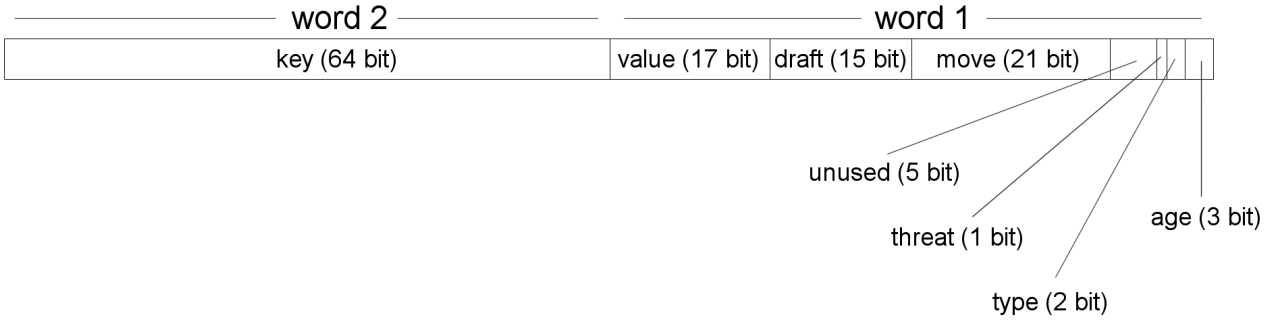
4.8 Tabelle delle trasposizioni

Anche Crafty usa tabelle hash per realizzare il meccanismo delle tabelle delle trasposizioni, in modo da salvare le posizioni della scacchiera valutate affinché possano tornare utili in futuro.

Una singola entry della tabella hash è definita in questo modo:

```
typedef struct {
   BITBOARD word1;
   BITBOARD word2;
} HASH_ENTRY;
```

per un totale di 128 bit. Vediamo qual è il significato di questi 128 bit.



- *key* è la chiave hash per accedere alla entry relativa,
- *value* è la valutazione della posizione,
- *draft* rappresenta la profondità della ricerca della posizione; serve per capire se è ragionevole usare il valore memorizzato,
- *move* è la mossa migliore calcolata a partire dalla posizione in esame al momento della memorizzazione del valore nella tabella,
- *threat* assume valore uno se la valutazione dà origine ad un extension,
- *type* dà informazioni su *value*: se è zero *value* è privo di significato, se è uno *value* è un lower bound nella ricerca AlphaBeta, due un upper bound e tre un valore esatto,
- *age* è un ID dato ad ogni entry che dipende dal momento in cui il valore è stato memorizzato.

Il campo *move* ha un utilizzo che per GNU Chess non è previsto: vi è memorizzata la mossa migliore (il nodo PV) e nelle ricerche successive, al momento dell'ordinamento dei nodi dell'albero, viene controllata per prima, nella speranza che causi un taglio.

Come in GNU Chess esiste anche una tabella per la struttura pedonale, costruita a partire dalla struttura `PAWN_HASH_ENTRY`. La dimensione della tabella hash normale è di 64KB, così come quella per i pedoni, suddivisa in 32 KB per i pezzi neri e 32KB per quelli bianchi.

Al momento della valutazione di una posizione si controlla se il punteggio è già presente nella tabella; ecco un esempio della valutazione della struttura pedonale:

```
ptable = pawn_hash_table + PawnHashKey;
if (ptable->key == PawnHashKey)           /* hash table hit */
   return score;
                                        /* hash table failure */
effettua_valutazione;
memorizza_punteggio_nella_tabella;
```

Vediamo ora come vengono inizializzate e aggiornate le chiavi hash. La funzione hash è la stessa di GNU Chess, ossia un'operazione di *xor*, anche se la tecnica di utilizzo di Crafty è un po' più complessa.

L'idea è questa: vengono utilizzati dei vettori random da 64 elementi, tanti quante le case della scacchiera. Ogni pezzo di ogni colore ha un vettore random. Quando è il momento di aggiornare la chiave si effettua uno *xor* tra il valore della chiave precedente e il valore del vettore random in corrispondenza della casa di partenza, quindi si effettua un ulteriore *xor* con il valore del vettore random in corrispondenza della casa di arrivo.

Vediamo l'esempio del cavallo nero.

La relativa chiave hash (già vista quando abbiamo esaminato la struttura `POSITION`) è

```
BITBOARD       hash_key;
```

mentre il suo vettore random è

```
BITBOARD       b_knight_random[64]
```

dove ogni elemento è di 64 bit.

L'inizializzazione del vettore è fatta nel seguente modo:

```
for (i = 0; i < 64; i++)
   b_knight_random[i] = Random64();
```

`Random64` è un generatore di numeri pseudo-casuali da 64 bit.

Al momento dell'aggiornamento della chiave, cioè immediatamente dopo aver mosso il cavallo nero dalla casa `from` alla casa `to`, vengono eseguite le istruzioni

```
HashNB(from,HashKey);
HashNB(to,HashKey);
```

dove `HashNB` è definita come segue:

```
#define HashNB(a,b)     b = b_knight_random[a]^(b)
```

In sostanza vengono eseguite le istruzioni

```
HashKey = b_knight_random[from]^HashKey;
HashKey = b_knight_random[to]^HashKey;
```

4.9 Le Tablebases

Le Tablebases sono database di finali che Crafty può utilizzare per giocare questa delicata fase della partita.

Per fare in modo che Crafty possa supportare questa funzionalità è sufficiente compilare l'eseguibile con il flag indicato (`–DEGTB6` per database di finali per un numero di pezzi fino a sei).

Una volta compilato correttamente l'eseguibile bisogna scaricare (gratuitamente) dalla rete le Tablebases. Ognuno di questi file rappresenta un finale, per esempio i due file `KBNK.tbb` e `KBNK.tbw` sono necessari per il finale re-alfiere-cavallo contro re.

Il problema è che questi file sono molto grandi: basti pensare che il set completo per i finali a quattro pezzi occupa in totale 260MB. Dei possibili finali a cinque pezzi

ne sono stati sviluppati finora solo cinque, ma si sta lavorando anche sugli altri. Solamente questi dieci file (ovvero le cinque coppie `.tbb` e `.tbw`) hanno una dimensione compressa di 500MB e di 2,3GB una volta scompattati.

Il programma è dotato di alcuni moduli per interfacciarsi correttamente con il database di finali.

Nella funzione `Search`, che realizza la ricerca nell'albero di mosse, si controlla la posizione sulla scacchiera (per ora Crafty non utilizza le Tablebases se non per un numero di pezzi minore o uguale a cinque). Qualora fossimo in presenza di un finale controlla se l'uso delle Tablebases è abilitato; dopo di che controlla se esiste la Tablebase relativa al finale in corso. Se anche questo controllo dà esito positivo inizia a giocare il finale sulla base delle mosse suggerite dalla Tablebase.

4.10 Libro di aperture

Crafty può venire dotato di un libro di aperture. Si tratta di un file PGN che deve essere compilato in un file `book.bin`.

I libri di aperture si possono scaricare gratuitamente dalla rete. A seconda dei limiti di spazio su disco e di velocità di download degli utenti, c'è la possibilità di scegliere tra tre diverse categorie di libri: *large* (suddiviso in quattro file `large1.zip`, `large2.zip`, `large3.zip` e `large4.zip`) che produce un database di aperture della dimensione di 60MB, *medium* (30MB) e *small* (1MB).

Vediamo come Crafty si interfaccia al libro di aperture. Prima di tutto esaminiamo le principali funzioni di *learning*, una tecnica che in GNU Chess non è presente.

Le funzioni di learning consentono al programma di valutare la bontà delle mosse scelte dal libro di aperture, analizzandone i risultati per capire se la mossa ha avuto successo o meno. I risultati di questa stima vengono registrati e usati in futuro per decidere se la mossa va riapplicata o meno. Per questo si dice che il programma *impara* dai propri successi o dai propri errori nell'utilizzo del libro.

Diamo uno sguardo alle principali funzioni che implementano questa particolare tecnica.

• `LearnBook`: esegue la valutazione delle prime `N` mosse effettuate fuori dal libro di aperture (`N` può essere variato). Dopo che queste `N` mosse sono state giocate la valutazione ci informa se l'ultima mossa presa dal libro era una buona o una cattiva mossa. Si possono presentare tre casi:

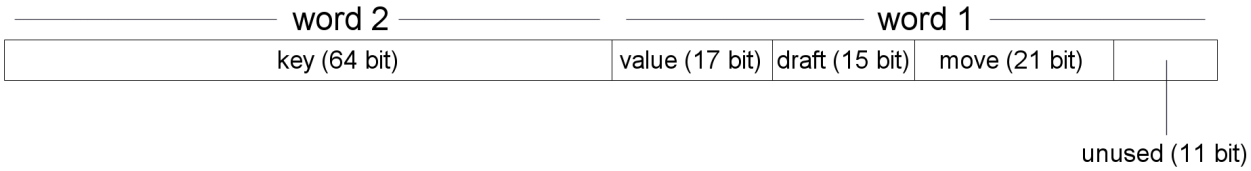
- se la mossa è considerata cattiva il relativo *learn value* viene decrementato,
- se la valutazione è ad un valore pari (né buona né cattiva) il learn value viene incrementato di poco, in modo tale da compensare più valutazioni pari con una cattiva,
- se infine la valutazione è buona il learn value viene incrementato sensibilmente.

Il learn value della mossa viene registrato in un file `book.lrn`, ma viene anche aggiornato nel libro di aperture, a questo serve la funzione `LearnBookUpdate`.

• Le funzioni `LearnImport` e `LearnImportBook` sono usate per leggere un file `*.lrn` e applicare i valori di learning al libro di aperture, eventualmente salvandoli e integrandoli al libro in formato binario.

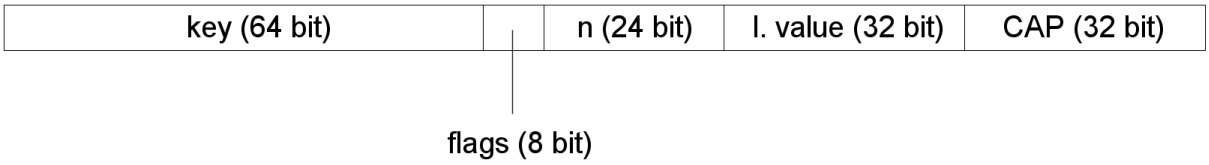
• `LearnPosition`: realizza un meccanismo di learning diverso dal precedente. Tale routine memorizza i risultati delle normali ricerche effettuate durante una partita in una tabella hash permanente, che viene mantenuta su disco. Quando inizia una nuova partita le informazioni contenute in essa vengono trasferite nella tabella delle trasposizioni attiva, facendo in modo che il programma inizi avendo una tabella delle trasposizioni ricca di informazioni già fin dai primi istanti della partita.

Il formato di una entry della tabella permanente è molto simile a quello della tabella delle trasposizioni.



Vediamo ora le caratteristiche del modulo `book`, che serve da interfaccia tra Crafty e il libro di aperture.

Il formato di una *book position* è il seguente:



- *key* è la chiave hash,
- i *flags* servono per arricchire la mossa con delle informazioni che potrebbero essere utili per decidere se la mossa va giocata o ignorata.
  - `0000 0001`: ?? (mossa molto cattiva, errore grave),
  - `0000 0010`: ? (mossa debole o probabile errore),
  - `0000 0100`: = (mossa che lascia immutata la situazione),
  - `0000 1000`: ! (mossa forte),
  - `0001 0000`: !! (mossa molto forte),
  - `0010 0000`: con questa mossa il nero ha vinto almeno una volta,
  - `0100 0000`: con questa mossa si è verificata almeno una patta,
  - `1000 0000`: con questa mossa il bianco ha vinto almeno una volta.
- `n` rappresenta il numero di partite in cui questa mossa è stata giocata,
- `l.value` è il valore di learning, visto prima,
- `CAP` è il *CAP score*, un ulteriore tipo di punteggio per l'uso del libro; non entriamo nei dettagli.

Guardiamo quali sono le principali funzioni del modulo `book`.

- `Book`: è usata per stabilire se una posizione fa match con una delle aperture presenti nel libro. Tale controllo viene effettuato controllando il valore `key`.

La funzione considera la lista di mosse generata a partire dalla posizione corrente, cioè a livello della radice, quindi le analizza tutte estrapolando il sottoinsieme di mosse presenti nel libro. Quindi cancella dalla lista le mosse che hanno il flag settato con ? o ??. Dalle mosse rimanenti cerca prima quelle con flag !!, poi quelle con flag !; se entrambe le ricerche falliscono sceglie una delle mosse rimanenti, eventualmente in maniera casuale.

• `BookPonderMove`: serve per scegliere una mossa su cui effettuare il pondering[2] senza ricorrere a una ricerca, ma semplicemente analizzando il libro e quindi scegliere come mossa da analizzare quella che risulta più probabile che l'avversario giochi, cioè quella con il contatore di frequenza più alto.

• `BookUp`: consente di creare un nuovo libro di aperture con il comando `book create`, oppure di aggiungere nuove mosse ad un libro già esistente mediante il comando `book add`.

Le mosse possono essere corredate dai caratteri ??, ?, =, !, !!, oltre che dalla dicitura `{play nn%}`, che forza il programma ad usare la mossa con una data frequenza percentuale.

4.11 Features

Ecco altre caratteristiche interessanti che Crafty offre.

_HISTORY DATABASE

È un database con entry da 12 bit, 6 per la casa di partenza, altri 6 per quella di arrivo, in cui si memorizzano le mosse più utilizzate, associate ad un contatore di frequenza. Al momento dell'ordinamento delle mosse si può eventualmente tenere conto dei suggerimenti dati dal database di history, analizzando per prime le mosse con il contatore più alto.

La stessa routine che gestisce l'history database si occupa anche della memorizzazione delle mosse killer, che hanno lo stesso formato di quelle history.

_ ANALYZE

Il comando `analyze` pone Crafty in un perenne stato di pondering. L'utente inserisce una mossa, il programma aggiorna lo stato della scacchiera e inizia il pondering di conseguenza, mostrando all'utente le analisi sull'albero delle mosse. Il programma non risponde alla mossa, ma attende che l'utente ne inserisca una successiva.

[2] Vedi par. 4.5.

_ ANNOTATE

Il comando `annotate` permette di rivisitare tutte le mosse di una partita, mentre Crafty le valuta una ad una, eventualmente commentandole e memorizzando l'output su un file con estensione `.can`, in formato PGN.

Il programma considera una data mossa; successivamente esegue una normale ricerca, come se stesse giocando una partita; quindi considera la mossa successiva e calcola lo scarto tra la valutazione della seconda mossa e la mossa che Crafty avrebbe scelto come conseguenza della sua ricerca. Se le mosse coincidono viene semplicemente memorizzata in formato PGN sul file; se le mosse hanno uno scarto di valutazione minore o uguale ad un certo margine viene memorizzata la mossa ed eventualmente stampata anche la mossa che avrebbe effettuato Crafty; se lo scarto supera il margine la mossa viene salvata unitamente ad un commento (della forma ??, !, =, += eccetera).

Il comando ha la forma

```
annotate filename b|w|bw|name moves margin time [n]
```

dove *`filename`* è il file di input. `b|w|bw|name` dice al programma che vanno analizzate soltanto le mosse del nero (`b`), del bianco (`w`), di entrambi (`bw`) o del giocatore `name`. `moves` indica quali mosse vanno esaminate; la mossa può essere una o più di una, è sufficiente indicare il range numerico; `1-999` significa che vogliamo studiare l'intera partita. `margin` è il margine di cui parlavamo prima: per dare un'idea il margine tra due valutazioni equivalente al valore di un pedone è `1.0`. `time` è il tempo concesso al programma per la ricerca della mossa ottima. Infine `[n]` è un parametro opzionale che permette di stampare, per ogni mossa analizzata, le prime `n` mosse migliori, complete del relativo punteggio.

Con il comando `annotateh` è possibile produrre l'output su un file HTML, completo di diagrammi in corrispondenza delle posizioni per cui il programma ha effettuato l'analisi.

_ EDIT

Permette di editare a piacere una posizione sulla scacchiera. Se è seguito dal carattere # pulisce completamente la scacchiera.

Una volta pulita la scacchiera è possibile sistemare i pezzi come si vuole digitando il comando `c` per stabilire il colore del pezzo da inserire, quindi il comando `[piece][square]`; per esempio `Qc5` pone una regina sulla casa *c5*. Infine `end` chiude la sessione di editing.

Naturalmente un tale uso del comando, cioè inserendo i comandi da tastiera, si rende necessario soltanto laddove non avessimo un'interfaccia grafica a disposizione. La funzione di editing è spesso già abilitata su un'interfaccia grafica come può essere Winboard/Xboard[3]; per esempio GNU Chess, di per sé, non supporta la funzione edit, ma se interfacciato con Winboard/Xboard è lo stesso possibile disporre di tale funzionalità grazie all'interfaccia grafica.

_ FUNZIONE DI TEST

Le funzioni di test servono per valutare le performance del programma se messo di fronte a delle posizioni stabilite dall'utente.

Il comando ha la forma `test` *`filename`*; il programma valuta la posizione contenuta nel file di input *`filename`* e successivamente effettua la ricerca della mossa migliore partendo da quella posizione. Eventualmente l'utente può inserire una lista di mosse che lui ritiene buone e il programma lo informa se la ricerca ha ritornato una mossa presente nella lista.

Come GNU Chess anche Crafty fornisce il supporto per la gestione di file in formato EPD e PGN. Consente inoltre l'utente di inserire le mosse in notazione algebrica estesa o in notazione SAN.

4.12 Alcuni comandi

Vediamo brevemente, così come abbiamo fatto per GNU Chess, alcuni comandi degni di attenzione che l'utente può impartire al programma.

- `alarm` abilita il segnale acustico associato all'inserimento di una mossa,
- `analyze` modalità *analyze*, vista nel paragrafo precedente,
- `annotate` funzione *annotate*, vista nel paragrafo precedente,
- `black (white)` dà la prima mossa al nero (bianco),

[3] Vedi cap. 6.

- `book` crea o aggiorna il libro di aperture, visto nel paragrafo 4.10,

- `cache` setta la dimensione della cache per le Tablebases (default 1MB),

- `display` visualizza la scacchiera, insieme ad una serie di informazioni che dipendono dai bit settati nella variabile `display_options`,

- `depth` setta la profondità massima che può raggiungere la ricerca,

- `draw` è usato quando l'utente vuole offrire a Crafty la patta,

- `easy (hard)` disabilita (abilita) il pondering,

- `edit` funzione di editing, vista nel paragrafo precedente,

- `egtb` abilita e disabilita l'uso delle Tablebases,

- `end` o `quit` termina il programma,

- `evaluation` usato per cambiare alcuni parametri della funzione di valutazione,

- `extensions` permette di decidere la profondità dei cammini delle extensions,

- `flip` inverte la scacchiera insieme al colore dei pezzi,

- `force` forza il programma a giocare una mossa specifica al posto di quella scelta a seguito della ricerca,

- `history` visualizza tutte le mosse fatte finora,

- `hash = nnn (nnnK) (nnnM)` setta la dimensione della tabella delle trasposizioni in byte (Kylobyte) (Megabyte),

- `hashp` setta la dimensione della tabella delle trasposizioni per i pedoni,

- `help` *`command`* visualizza un aiuto per il comando *`command`*,

- `hint` visualizza la migliore mossa che il programma si aspetta,

- `ics` permette di settare i parametri da usare nel dialogo con l'Internet Chess Server,

- `import` *`file.lrn`* serve per integrare *`file`*`.lrn` nel relativo libro in formato `.bin`,

- `learn` abilita/disabilita il meccanismo di learning,

- `level` setta il controllo del tempo,

- `list` permette di gestire (aggiungere/rimuovere) nomi di giocatori,

- `log` fa iniziare/terminare la sessione di log,

- `name` permette di inserire il nome del giocatore e di salvarlo in un file di log,

- `new` fa ripartire il programma con un nuova partita,

- `output` sceglie tra la notazione estesa e ridotta per l'output,

- `ponder` attiva/disattiva il pondering oppure richiede il servizio su una particolare mossa,

- `post (nopost)` attiva (disattiva) il display del thinking,

- `remove` annulla le ultime due semimosse,

- `read` consente di inserire una lista di mosse da tastiera oppure da file,

- `resign` è usato dall'utente per abbandonare la gara,

- `savegame` salva la partita in un file PGN,

- `savepos` salva la partita in un file secondo la notazione FEN[4],

- `selective` setta la profondità minima e massima per l'utilizzo di mosse nulle,

- `score` visualizza la valutazione di una particolare posizione,

- `sd` setta una specifica profondità per l'albero di ricerca,

- `smpin` è usato per stabilire la profondità minima per cui un thread può essere attivato,

- `smpmt` consente di decidere il numero massimo di thread,

- `sn` stabilisce un numero massimo di nodi da visitare prima che la ricerca termini,

- `st` setta un tempo specifico per la ricerca,

- `store` memorizza la posizione corrente su file `position.bin`,

- `test` esegue la funzione di test, vista nel paragrafo precedente,

- `time` controlla il tempo per la ricerca (è possibile specificare l'incremento per ogni mossa),

- `undo` annulla l'ultima semimossa,

- `xboard` setta i parametri per l'uso dell'interfaccia Xboard.

---

[4] La notazione FEN (Forsythe-Edwards Notation) è una notazione che descrive lo stato della scacchiera. Ogni stringa, letta da sinistra verso destra, rappresenta la scacchiera. Le lettere maiuscole rappresentano i pezzi bianchi, quelle minuscole i neri. I numeri rappresentano le case vuote consecutive, mentre la barra / indica la fine di una traversa dopo che tutti i pezzi sono stati inseriti. Alla fine della stringa ci sono altri caratteri indicanti il lato che ha il tratto, lo stato degli arrocchi e delle catture en passant.
Per esempio, la stringa `K2R/PPP////q/5ppp/7k/ b - -` rappresenta la seguente posizione:

```
K  -  -  R  -  -  -  -
P  P  P  -  -  -  -  -
-  -  -  -  -  -  -  -
-  -  -  -  -  -  -  -
-  -  -  -  -  -  -  -
q  -  -  -  -  -  -  -
-  -  -  -  -  p  p  p
-  -  -  -  -  -  -  k
```

Da notare il fatto che mentre GNU Chess non supporta il dialogo con un Internet Chess Server se non attraverso l'interfaccia Winboard/Xboard, Crafty è abilitato al dialogo con il server anche senza l'intermediazione dell'interfaccia, con apposite routine per convertire le mosse nel formato utilizzato dal server e per leggere le mosse in arrivo dal server e convertirle nel formato interno al programma.

5 Interfacce grafiche

5.1 Cos’è un’interfaccia grafica per un programma di scacchi

Per giocare a scacchi contro un computer è indispensabile utilizzare un’interfaccia. Con il termine interfaccia non si intende necessariamente un software grafico, ma semplicemente il modo con cui il programma dialoga con l’utente, prende le mosse in input e visualizza un output, cioè le mosse con cui il programma risponde e lo stato della scacchiera.

Per esempio, un’interfaccia di un motore scacchistico che non usa un software a parte per visualizzare la scacchiera, ma che dispone unicamente della possibilità di stampare caratteri a video, può essere qualcosa del genere:

```
---------------------------------
| R | N | B | Q | K | B | N | R | 8
---------------------------------
| P | P | P | P | P | P | P | P | 7
---------------------------------
|   |   |   |   |   |   |   |   | 6
---------------------------------
|   |   |   |   |   |   |   |   | 5
---------------------------------
|   |   |   |   |   |   |   |   | 4
---------------------------------
|   |   |   |   |   |   |   |   | 3
---------------------------------
| p | p | p | p | p | p | p | p | 2
---------------------------------
| r | n | b | q | k | b | n | r | 1
---------------------------------
  a   b   c   d   e   f   g   h

your move:_
```

Naturalmente, se un tempo questo era l’unico metodo per la visualizzazione, col passare degli anni sono state sviluppate numerose interfacce grafiche, più gradevoli e più comode, che permettono di inserire le mosse semplicemente spostando il pezzo col mouse.

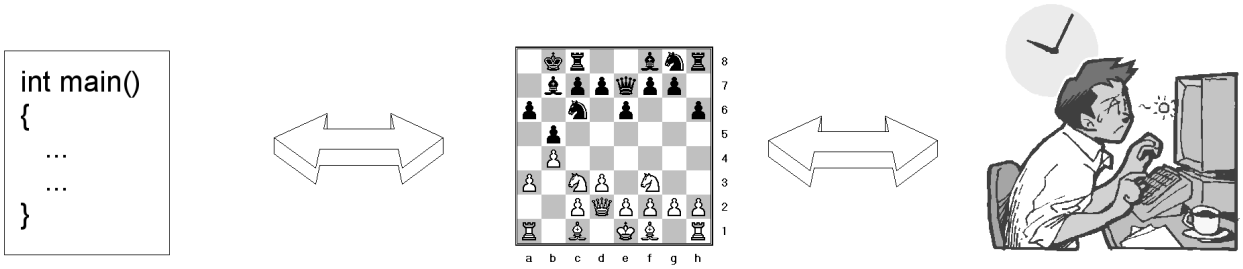
Le interfacce grafiche sono un software a parte; chi sviluppa un programma per scacchi non deve dare per scontato che l’utente ne potrà disporre, ed è quindi suo dovere dotare il motore di un’interfaccia rudimentale ma comprensibile, come quella della figura precedente.

5.2 Come funziona

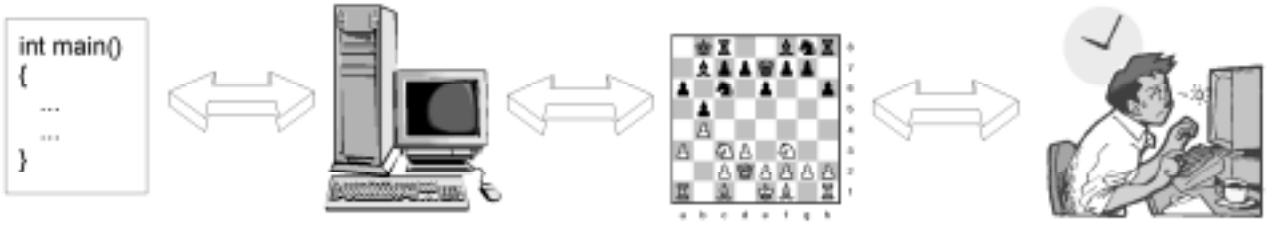
Esistono diverse interfacce grafiche, ognuna con le sue caratteristiche e ognuna con le sue tecniche. È quindi arduo rispondere in modo univoco alla domanda “come funzionano?”.

Innanzitutto cominciamo col definire i ruoli di un’interfaccia grafica.

Un’interfaccia può consentire il dialogo tra un essere umano e un programma;



ma anche fungere da client per un Internet Chess Server[1] e quindi mediare tra server ed essere umano.



Un simile software dovrà quindi avere degli strumenti per comunicare con il motore scacchistico sottostante: per prendere cioè in input la mossa scelta dal motore e per comunicare la mossa effettuata dall’essere umano, inoltre dovrà essere in grado di scambiare tutta una serie di informazioni riguardanti l’inizializzazione della partita, lo stato del gioco eccetera.

Se la nostra interfaccia è in grado di fungere da client verso un Internet Chess Server dovrà anche avere la possibilità di comunicare con quest’ultimo: ciò si traduce nell’apertura di una connessione, nella ricezione dei dati, nell’estrazione delle informazioni dai dati (tra cui le mosse) e l’invio delle mosse dell’utente.

[1] Vedi cap. 7.

5.3 I servizi offerti

Oltre a consentire il dialogo con il programma di scacchi ed eventualmente con l'Internet Chess Server (vedremo questo caso più in dettaglio nei capitoli 7 e 8, esaminando anche le funzionalità proprie dei server scacchistici, che tali interfacce mettono a disposizione), le interfacce grafiche per scacchi offrono solitamente dei servizi standard, tra cui:

- la possibilità di salvare e caricare partite (in formato PGN o simili),
- salvare e caricare posizioni (in formato EPD, FEN o simili),
- editare a piacere posizioni,
- corredare le partite salvate di commenti (in linguaggio naturale) e di simboli (!, =, ??, …),
- impostare a piacere suoni, colori, fonts,…

e altri ancora.

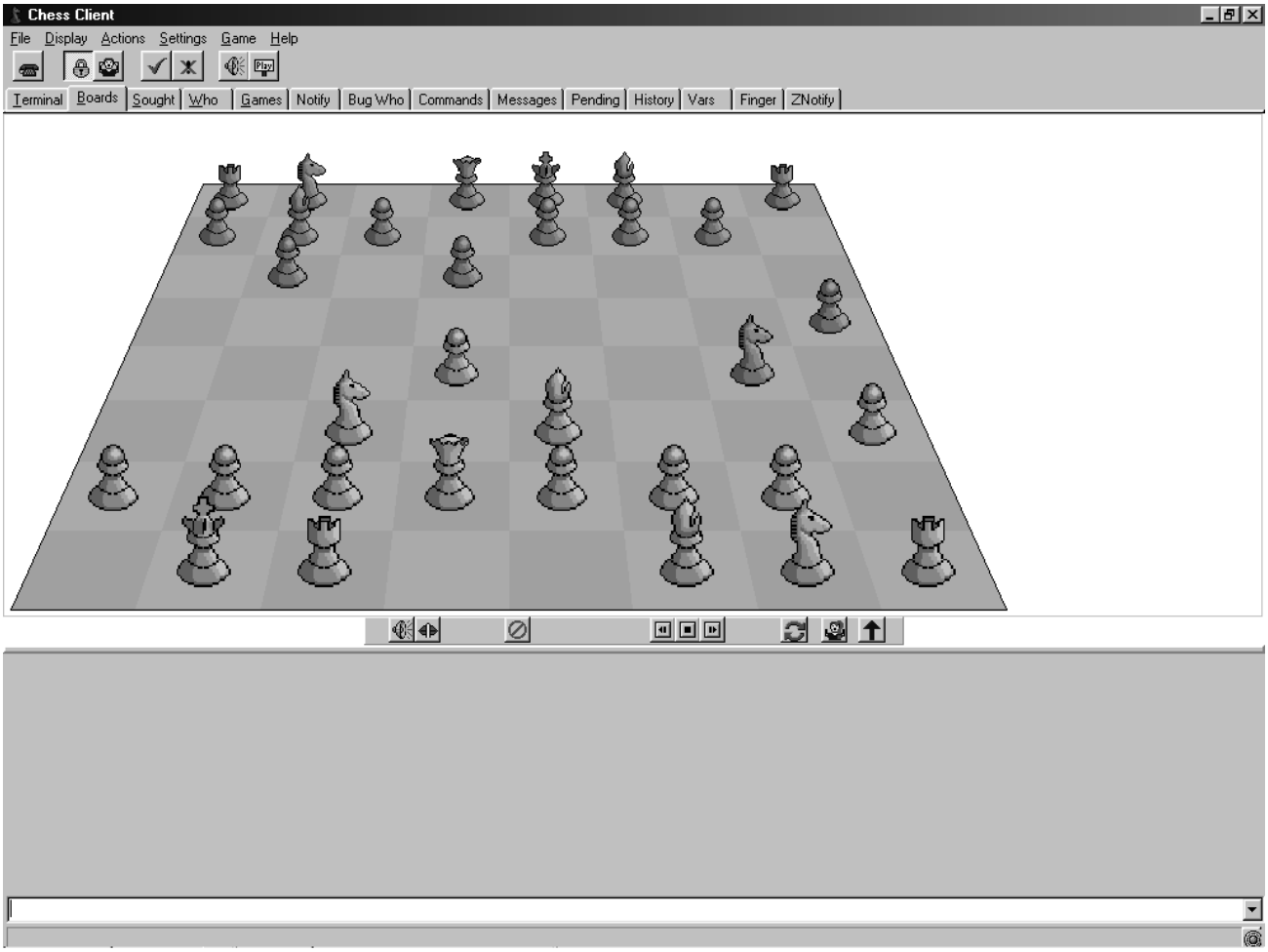
5.4 Alcune interfacce grafiche

L'interfaccia in assoluto più usata per i programmi di scacchi che circolano sulla rete è Winboard/Xboard, che vedremo approfonditamente nel capitolo successivo. Esistono però altri tipi di interfaccia. Troviamo altre interfacce generiche, come Winboard/Xboard, cioè scaricabili ovunque e compatibili con più programmi per scacchi, sia freeware che commerciali, e abbiamo poi le interfacce proprie di ogni Internet Chess Server. Spesso, infatti, accade che l'ICS metta a disposizione un'interfaccia scaricabile gratuitamente dal sito stesso, che consenta agli utenti di effettuare il login con il server e giocare on-line. Alcuni server più poveri mettono a disposizione semplici interfacce realizzate con Applet Java oppure attraverso pagine ASP e quindi visualizzabili con appositi browser. L'unico inconveniente è che ogni volta che ci si connette al server bisogna aspettare che la pagina contenente l'applet si ricarichi.

Questi sono alcuni esempi del primo tipo di interfaccia.

Π *CClient*

È un'interfaccia per piattaforma Windows, consente di effettuare il login ad un ICS e offre un'ampia gamma di servizi, tra cui la possibilità di scegliere tra visione 2D e 3D della scacchiera. È scaricabile dal sito *http://home.centurytel.net/khb/cclient*



Π *ChessMachine*

Per ambiente Windows, scaricabile dal sito *http://www-abc.mpib-berlin.mpg.de/users/dudey/ChessMachine093.exe*

Π *Freak*

Per Windows. *ftp://ftp.freechess.org/pub/chess/Windows/freak/Freak.exe*

Π *LightningRulez*

Per Windows. *http://gilly.homeip.net/lr/*

Π *SLICS*

Per ambiente Windows. *http://www.dfong.com/chessbd/*

Π *Thief*

Per Windows. *http://www.thebugboard.net/downloads.htm*

Π *E-ICS*

Per MacOs. *ftp://ftp.freechess.org/pub/chess/Macintosh/E-ICSv0.93.sea.hqx*

Π *Fixation*

Per MacOs. *http://www3.sk.sympatico.ca/cknelsen/fixation/fixation.html*

Π *MacICS-TCP*

Per MacOS. Scaricabile da *ftp://ftp.freechess.org/pub/chess/Macintosh/MacICS-TCPvFICS.sit.hqx*

Π *PowerICS*
Per MacOS. *ftp://ftp.freechess.org/pub/chess/Macintosh/PowerICS_Z-1.5.hqx*

Π *Eboard*
Per sistema operativo Unix/Linux. *http://eboard.sourceforge.net/*

Π *Xics*
Per Unix/Linux. *ftp://ftp.freechess.org/pub/chess/X/xics-2.3.tar.gz*

Π *AmyBoard*
Per Amiga OS. *ftp://ftp.freechess.org/pub/chess/Amiga/AmyBoard330.5bin.lha*

Π *PMICS*
Per OS/2. *ftp://ftp.freechess.org/pub/chess/OS2/pmics211.zip*

Π *Bludrake*
Per Windows 3.1. *ftp://ftp.freechess.org/pub/chess/Windows/bludrake1.2.zip*

Π *Gilchess*
Per Windows 3.1. *ftp://ftp.freechess.org/pub/chess/Windows/gilchess7.zip*

Π *Raja*
Per Windows 3.1. *ftp://ftp.freechess.org/pub/chess/Windows/raja40.zip*

Π *SLICS*
Per Windows 3.1. *http://www.dfong.com/chessbd/*

Π *Toolkit*
Per Windows 3.1. *ftp://ftp.freechess.org/pub/chess/Windows/tolk13.exe*

Π *GIICS-v303*
Per MS-DOS. *ftp://ftp.freechess.org/pub/chess/DOS/giics303.exe*

Π *JIICS*
Per MS-DOS. *ftp://ftp.freechess.org/pub/chess/DOS/jiics120.exe*

Π *NGIICS*
Per MS-DOS. *ftp://ftp.freechess.org/pub/chess/DOS/ngiics303.exe*

Π *Monarc*
Per MS-DOS. *ftp://ftp.freechess.org/pub/chess/DOS/monarch2.zip*

Π *ZIICS-v130*
Per MS-DOS. *ftp://ftp.freechess.org/pub/chess/DOS/ziics131.exe*

Le due interfacce più utilizzate, assieme a Winboard/Xboard, sono Chessbase (*www.chessbase.com/*) e Shredder (*www.computerchess.com/index_e.html*). Queste tre interfacce devono la loro popolarità al fatto che, utilizzando il medesimo protocollo, garantiscono che la maggior parte dei programmi compatibili per l'una sia anche compatibile per l'altra. La differenza sostanziale tra Winboard/Xboard, Chessbase e Shredder è che mentre la prima è freeware le altre due sono prodotti commerciali.

6 Winboard/Xboard

6.1 Aspetti generali

Creata da Tim Mann (*www.tim-mann.org/chess.html*) l'interfaccia grafica Winboard/Xboard è ormai un punto di riferimento per tutti i giocatori di scacchi in rete. Si può scaricare da molti siti, uno tra i quali è proprio quello di Tim Mann.

Winboard è il software compatibile con MS Windows 9x/NT, mentre Xboard è stata scritta per il sistema operativo Linux. Winboard/Xboard viene distribuita unitamente al motore scacchistico GNU Chess, versione 4; le ultime versioni includono anche la versione 5.

Perché questa interfaccia ha avuto un così largo consenso? Soprattutto per la compatibilità con numerosi motori per scacchi, fatto che ha portato in breve la gran parte degli appassionati ad adottarla come standard grafico e a sviluppare nuovi programmi tutti compatibili con Winboard/Xboard, aumentando in modo sempre più crescente il grado di compatibilità fra l'interfaccia e i motori freeware presenti in rete.

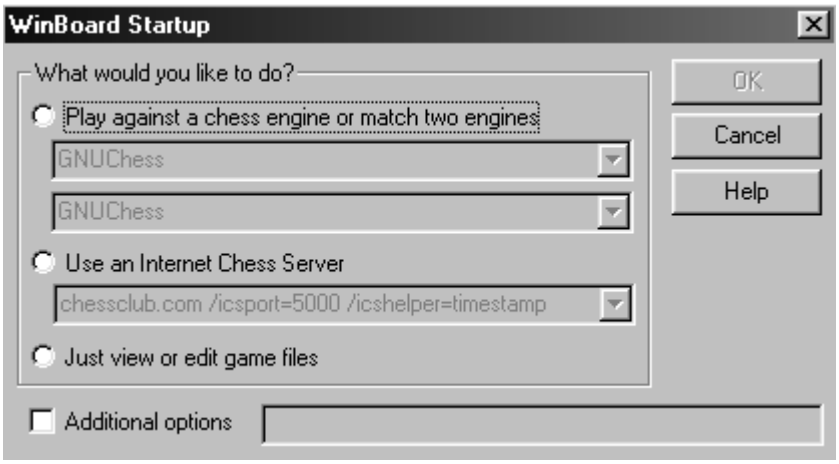
Tale successo ha fatto sì che si creassero comunità di giocatori e programmatori che utilizzano esclusivamente Winboard/Xboard, organizzando anche tornei per motori scacchistici che usano questa interfaccia.

A titolo di esempio menzioniamo l'ultimo torneo Winboard (2001), che ha visto scontrarsi 10 motori scacchistici su un computer dotato di CPU Amd K7 Athlon 1,0 Ghz con 128 MB di RAM e sistema operativo Microsoft Windows 98 SE. Il torneo è stato giocato con interfaccia Winboard versione 4.2.3 ed è stato vinto da Crafty versione 18.1.

La versione di Winboard/Xboard descritta in questo capitolo è la 4.2.5.

6.2 Funzioni

Il primo messaggio che ci appare quando avviamo Winboard[1] è questo,

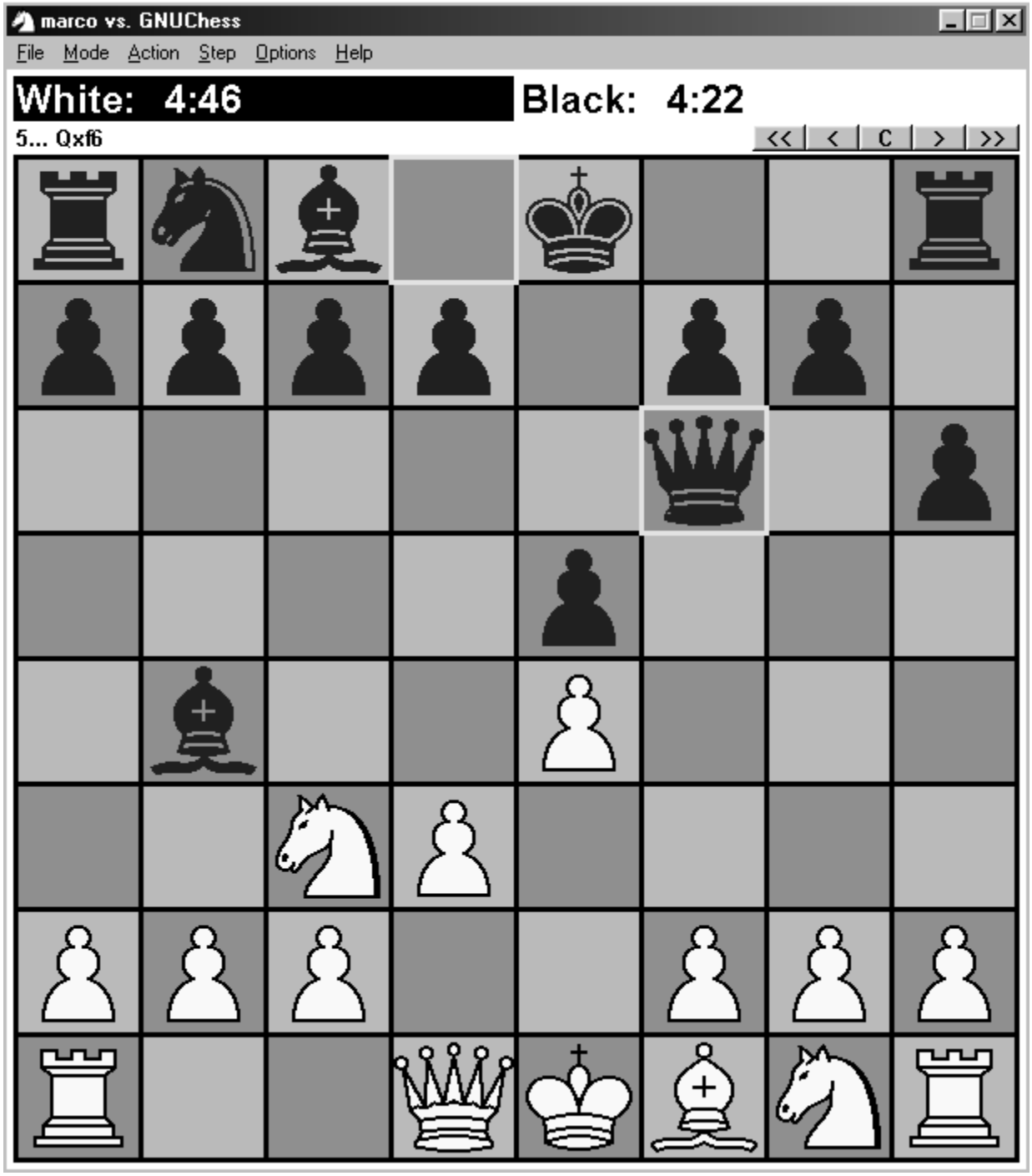


questa finestra ci fa immediatamente capire quali sono le tre grandi categorie di servizi che l'interfaccia offre:

- giocare contro un motore scacchistico oppure fare in modo che due motori si sfidino tra di loro,
- giocare su un Internet Chess Server,
- editare, salvare e caricare file in formato standard.

La scacchiera si presenta all'utente in questa veste grafica:

[1] Con Xboard, sotto il sistema operativo Linux, una tale finestra non c'è; comunque è possibile effettuare la scelta tra le tre opzioni con una semplice riga di comando.



che può comunque essere modificata a piacere. Con il menu a tendina *Options* è infatti possibile cambiare i colori delle case e dei pezzi, nonché le dimensioni degli oggetti e della finestra; è altresì possibile cambiare i fonts e impostare i suoni preferiti.

Sempre dal menu *Options* si accede ad una funzione chiamata *Time control*, tramite la quale si possono impostare i parametri di tempo da usare durante la gara, compresi eventuali incrementi per ogni mossa.

Winboard/Xboard consente di salvare e leggere partite in formato PGN, nonché posizioni in formato EPD e FEN.

Il menu *Mode* consente di

- stabilire il colore con cui gioca il motore,
- scegliere di far scontrare due motori di scacchi tra di loro,
- attivare la funzione *analyze* (solo se il programma sottostante la supporta),
- editare partite e posizioni,
- editare tag (??, !, =+, …) e commenti.

Dal menu *Action* si possono impartire dei comandi relativi alla gara in corso, tra i quali alcuni abilitati soltanto se si sta usando un ICS (*Accept*, *Recline*, *Rematch*, *Stop Observing*, *Stop Examining*). I comandi *Call Flag* (nel caso l'avversario abbia ecceduto nel tempo), *Abort*, *Draw* e *Resign*, invece, sono azionabili anche in modalità off-line.

Il menu *Step* si riferisce invece alla sequenza di mosse della partita. Il comando *Type In Move* permette di inserire una mossa manualmente (es.: *e2e4*): può essere usato quando si vuole inviare una mossa ad un ICS superando il test di legalità effettuato dall'interfaccia. Abbiamo poi i comandi *Backward*, *Forward*, *Back To Start*, *Forward To End* che servono per navigare attraverso la sequenza di mosse effettuate finora, *Move Now*, che forza il programma ad effettuare la mossa prima che la ricerca abbia termine, e *Retract Move*, per effettuare l'undo dell'ultima mossa.

6.3 Winboard/Xboard e motori scacchistici

In questo paragrafo vedremo come Winboard/Xboard si interfaccia con un motore per scacchi.

Innanzitutto vediamo come si fa per lanciare Winboard/Xboard con un motore scacchistico diverso da GNU Chess.

Nel caso di Winboard è sufficiente modificare il file `Winboard.ini` contenuto nella directory dove è stato installato il programma. Alla fine del file ci sono le istruzioni che informano l'interfaccia di quali siano il primo e il secondo motore scacchistico in uso (questo non vuol dire che i motori possono essere solo due, `firstChessProgam` e `secondChessProgram` devono avere entrambi almeno un

valore, che può anche coincidere, perché sia possibile far giocare due motori fra di loro);

```
/firstChessProgramNames={GNUChess
“GNUChes5 xboard”
}
/secondChessProgramNames={GNUChess
“GNUChes5 xboard”
}
```

A questa porzione di file possiamo ad esempio sostituire la seguente

```
/firstChessProgramNames={GNUChess
“GNUChes5 xboard”
“Crafty” /fd=”c:\Programs\ChessPrograms\Crafty18.9”
}
/secondChessProgramNames={GNUChess
“GNUChes5 xboard”
“Crafty” /sd=”c:\Programs\ChessPrograms\Crafty18.9”
}
```

dove `Crafty18.9` è la directory che contiene l’eseguibile di Crafty. Una volta salvato il file e eseguito nuovamente il programma, sia nella lista relativa al primo motore, che in quella relativa al secondo motore, compariranno sia GNU Chess che Crafty.

Col sistema operativo Linux, invece, supponendo di voler giocare con Crafty come primo motore, lanciamo Xboard con i seguenti parametri:

```
xboard -fcp "./crafty" -fd crafty_directory
```

6.3.1 Come avviene la comunicazione: pipe

Winboard/Xboard viene eseguito separatamente dal motore di scacchi: sono due processi diversi. Questi due processi, sia con Xboard che (contrariamente a quanto si potrebbe pensare) con Winboard, comunicano attraverso una coppia di pipe, una usata per l’input e l’altra per l’output.

Il motore di scacchi non è a conoscenza di queste pipe, che vengono create, inizializzate e gestite da Winboard/Xboard. Vediamo la funzione che crea le pipe;

```
void SetUpChildIO(to_prog, from_prog)
   int to_prog[2], from_prog[2];
{
   signal(SIGPIPE, SIG_IGN);
   pipe(to_prog);
   pipe(from_prog);
}
```

Tra interfaccia e programma si realizza una comunicazione client-server; il processo figlio viene creato in questo modo:

```
int StartChildProcess(cmdLine, dir, pr)
{
   ...
   int to_prog[2], from_prog[2];
   ...
   SetUpChildIO(to_prog, from_prog);

   if ((pid = fork()) == 0) {
     /* Child process */
     dup2(to_prog[0], 0);
     dup2(from_prog[1], 1);
     close(to_prog[0]);
     close(to_prog[1]);
     close(from_prog[0]);
     close(from_prog[1]);
     ...
     }

   /* Parent process */
   close(to_prog[0]);
   close(from_prog[1]);
   ...
   return 0;
}
```

Le mosse e i dati vengono infine inviati al programma attraverso la semplice istruzione

```
OutputToProcess(pr, message, count, &error);
```

vengono invece semplicemente visualizzati a video per l’essere umano con il comando

```
OutputToProcess(NoProc, data, length, &error);
```

La differenza importante tra le due istruzioni è il primo parametro, che permette di capire chi è il destinatario del messaggio. Questo è ciò che esegue la routine `OutputToProcess` al suo interno:

```
ChildProc *cp = (ChildProc *) pr;

if (pr == NoProc)                                  /* to human */
   fwrite(message, 1, count, stdout);
else                                          /* to chess engine */
   write(cp->fdTo, message, count);
```

6.3.2 Segnali

Oltre che attraverso pipe l'interfaccia dialoga con il programma anche per mezzo dell'invio di signal. Naturalmente questo accade soltanto per Xboard, dal momento che Windows non possiede segnali Unix-style.

Quando viene impartito il comando *quit*, Xboard invia al motore un signal `SIGTERM`, per essere sicura che il programma sia realmente terminato.

Il segnale `SIGINT`, invece, viene mandato in certi casi quando il motore non è in attesa di un input da parte dell'utente (per esempio nei casi di thinking o pondering), oppure quando viene forzata la mossa al programma (tramite un comando di *Move Now*).

6.3.3 Messaggi da Winboard/Xboard al programma

Vediamo i messaggi che Winboard/Xboard può inviare al motore per scacchi. I comandi che non commenteremo sono quelli già visti in precedenza (per GNU Chess o per Crafty), le cui caratteristiche dovrebbero già risultare chiare.

- `xboard` mandato subito dopo che il processo è partito per porre il motore in *xboard mode*,
- `protover` *`N`* se *`N`* `= 2` significa che stiamo usando la seconda versione del protocollo di Xboard, se la stringa non arriva stiamo usando la prima versione,
- `accepted (rejected)` mandato per accettare o rifiutare il settaggio di una feature (vedi seguito) da parte del programma,
- `new` resetta la scacchiera e dà la mossa al bianco,
- `variant` *`NAME`* per scegliere una variante per scacchi,

- `quit`,
- `random` serve solo a GNU Chess, usato per attivare/disattivare la funzione random nella valutazione delle posizioni,
- `force`,
- `go` esce dal *force mode*,
- `white (black)`,
- `level` *`MPS BASE INC`* o `st` *`TIME`*,
- `sd` *`DEPTH`*,
- *`MOVE`* inviato al momento di mandare una mossa al motore; a meno che la features `san` non sia attivata (vedi seguito); WinBoard/Xboard invia le mosse al programma usando la notazione CAN, per esempio:

Normal moves:	e2e4
Pawn promotion:	e7e8q
Castling:	e1g1, e1c1, e8g8, e8c8

- `?` è il comando *Move Now*, già visto quando abbiamo presentato le caratteristiche dell'interfaccia grafica,
- `result` *`RESULT`* {*`COMMENT`*} inviato alla fine di ogni partita; esempio:

```
result 1-0 {White mates}
```

- `setboard` *`FEN`* attivato soltanto da una features da parte del programma; permette di settare una posizione sulla scacchiera,
- `edit`,
- `hint`,
- `undo`,
- `remove`,
- `hard (easy)`,
- `post (nopost)`,
- `analyze`,
- `name` *`NAME`*,
- `rating` usato su ICS quando si ottiene il rating dell'avversario,
- `ics` *`HOSTNAME`* comunica che si sta giocando sul server `HOSTNAME`,
- `computer` informa il programma che il suo avversario è un altro programma,

- `pause (resume)` pone (risveglia) il programma in (da) uno stato di attesa.

6.3.4 Messaggi dal programma a Winboard/Xboard

- `feature` *`FEATURE1=VALUE1 FEATURE2=VALUE2…`* usato dal programma per scegliere alcune opzioni, vediamo soltanto qualche esempio:

  - `san` se `san=1` Winboard/Xboard invierà le mosse in notazione SAN,
  - `draw` se `draw=0` Winboard/Xboard non informerà il programma delle offerte di patta,
  - `sigint` se `sigint=0` la possibilità dell'invio di un signal SIGINT è disabilitata,
  - `sigterm` se `sigterm=0` la possibilità dell'invio di un signal SIGTERM è disabilitata,
  - `analyze` se `analyze=0` Winboard/Xboard non invia il comando `analyze` al programma ma visualizza subito un messaggio d'errore,
  - `variants` informa Winboard/Xboard delle varianti che il programma accetta,

- `Illegal move: MOVE` o `Illegal move (REASON): MOVE` Winboard/Xboard non effettua un controllo completo di legalità sulle mosse. Per esempio non riconosce se un arrocco non è più valido perché una torre o il re sono stati mossi, nemmeno è in grado di valutare se una cattura en passant sia valida.

Per questo potrebbe capitare che al motore scacchistico giungano delle mosse illegali. È compito del programma rispondere a Winboard/Xboard con un messaggio d'errore, in modo che la mossa possa essere ritratta. Ecco qualche esempio:

```
Illegal move: e2e4
Illegal move (in check): Nf3
Illegal move (moving into check): e1g1
```

- `Error (`*`ERRORTYPE`*`):` *`COMMAND`* usato dal programma laddove riceva un comando errato o incomprensibile. Per esempio:

```
Error (ambiguous move): Nf3
Error (unknown command): analyze
Error (command not legal now): undo
Error (too many parameters): level 1 2 3 4 5 6 7
```

- `move` *`MOVE`* per inviare la mossa in formato algebrico esteso o SAN,

- *RESULT* {*COMMENT*},
- resign,
- offer draw,
- tellopponent *MESSAGE* invia un messaggio all'avversario (funziona sia in locale che su ICS).

6.4 Winboard/Xboard e ICS

Winboard/Xboard è in grado di fungere da client con un Internet Chess Server e dare quindi la possibilità all'essere umano di usare tale interfaccia per giocare contro altri esseri umani o altri programmi in remoto.

Per usare Winboard con un ICS ci sono tre modi:

a) cliccare sull'opzione *Use an Internet Chess Server* nella finestra di startup[2], scegliendo dalla lista l'ICS preferito,

b) cliccare sull'opzione *Use an Internet Chess Server*, aggiungendo le opzioni desiderate nel campo *Additional Options*,

c) cliccando col tasto destro sull'icona di Winboard, scegliere *Proprietà* e aggiungere alla riga di comando contenuta nel campo *Destinazione* il comando completo delle opzioni desiderate per l'uso dell'ICS.

Con Xboard, invece, si procede digitando un'opportuna riga di comando;

```
xboard –ics     oppure     xboard –internetChessServerMode True
```

Questo comando, per default, tenta di connettersi a chessclub.com[3]; per scegliere un host diverso si digita

```
xboard –ics –icshost hostname –icsport portnumber
```

Naturalmente ci sono moltissime altre opzioni che si possono scegliere; si rimanda alla documentazione del software per un elenco completo.

[2] Vedi paragrafo 6.2.
[3] È un Internet Chess Server. Vedi capitolo 7 e 8.

6.4.1 Stabilire la connessione

La connessione può essere stabilita sia via Telnet che via TCP. Diamo uno sguardo a come avviene.

```
int establish()
{
   ...
   if (useTelnet)
      return OpenTelnet(icsHost, icsPort, &icsProcess);
   else
      return OpenTCP(icsHost, icsPort, &icsProcess);
}
```

dove `icsHost` e `icsPort` contengono rispettivamente i valori “`chessclub.com`” e “`5000`”.

Seguono i dettagli più interessanti della funzione `OpenTelnet`.

```
int OpenTelnet(host, port, pr)
   char *host; char *port; ProcRef *pr;
{
   ...
   char cmdLine[MSG_SIZ];

   sprintf(cmdLine, "%s %s %s", telnetProgram, host, port);
   return StartChildProcess(cmdLine, "", pr);
}
```

dove `telnetProgram` contiene il valore `telnet`.

La routine non fa altro che creare un nuovo processo che eseguirà, con un’istruzione `exec`, l’istruzione contenuta in `cmdLine`, cioè

```
telnet chessclub.com 5000
```

Diamo un’occhiata ad alcune istruzioni della routine `OpenTCP`.

```
int OpenTCP(host, port, pr)
   char *host; char *port; ProcRef *pr;
{
   int s;
```

```
    struct sockaddr_in sa;
    ...
    if ((s = socket(AF_INET, SOCK_STREAM, 6)) < 0)
       return errno;

    memset((char *) &sa, (int)0, sizeof(struct sockaddr_in));
    sa.sin_family = AF_INET;
    sa.sin_addr.s_addr = INADDR_ANY;
    ...
    if (bind(s, (struct sockaddr *) &sa, sizeof(struct
          sockaddr_in)) < 0)
       return errno;
    ...
    if (connect(s, (struct sockaddr *) &sa, sizeof(struct
          sockaddr_in)) < 0) {
       return errno;
    }
    ...
    return 0;
 }
```

Ci sono poi numerose funzioni che vengono utilizzate per il dialogo vero e proprio con il server, come ad esempio `ReadFromICS`, `SendToICS` e `SendMoveToICS`.

6.4.2 Il parsing dell'output dell'ICS

Quando Winboard/Xboard agisce come client di un ICS deve essere in grado di capire i messaggi che giungono dal server per poterli trasmettere al programma, all'utente o comunque per eseguire delle azioni che diventano necessarie qualora si riceva un determinato messaggio.

Il metodo usato da Winboard/Xboard è molto semplice: attraverso la funzione `looking_at` si confronta il buffer dove viene memorizzato l'output del server con una lista di possibili pattern, nella speranza che uno di questi faccia match con la stringa contenuta nel buffer. Se viene trovato un pattern che combacia, i caratteri nel buffer che sono stati riconosciuti vengono consumati.

In un pattern i caratteri devono corrispondere esattamente, tranne il carattere "`*`", che fa match con qualsiasi sequenza di zero o più caratteri, escluso il newline.

Vediamo un piccolo sottoinsieme di questi pattern.

```
"\"*\" is *a registered name"
"Logging you in as \"*\""
"Your name will be \"*\""
```

```
"* s-shouts: "
"* shouts: "
"* tells you: "
"* says: "
"* has left a message "
"*. * (*:*): "
"*. * at *:*: "
"* whispers: "
"*)(*): *"
"*(*): *"
"*)(*)(*): *"
"Challenge:"
"* offers you"
"* requests to"
"login:"
"* * match, initial time: * minute*, increment: * second"
"Move  "
"Adding game * to observation list"
"Game notification: * (*) vs. * (*)"
"Illegal move"
"Your king is in check"
"It is not your move"
"Removing game * from observation"
"no longer observing game *"
"\n"
```

L’ultimo pattern, `\n`, consente di eliminare tutti i caratteri del buffer fino al prossimo newline: serve qualora la stringa in questione non abbia fatto match con nessuno dei pattern precedenti.

6.5 Zippy

Zippy è un piccolo programma creato da Tim Mann nel 1993, che permette ad un motore scacchistico compatibile di giocare su un Internet Chess Server. Non è nient’altro che del codice aggiuntivo che definisce nuove routine oppure che utilizza routine proprie di Winboard/Xboard. Zippy è incluso nelle distribuzioni sia di Winboard che di Xboard.

La documentazione del programma sottolinea più volte questo fatto: nel caso si desideri far giocare un programma su un Internet Chess Server, a meno che non si entri come *guest* per effettuare qualche piccolo esperimento, è dovere del programmatore avvertire gli amministratori del server che il giocatore è una macchina (viene aggiunta la dicitura `(c)` dopo il nome del giocatore). Questo per evitare che qualcuno usi un programma forte per giocare in rete, vedendo il proprio rating salire senza effettivamente meritare il punteggio.

In ambiente Unix/Linux, per abilitare l'uso di tale programma, bisogna compilare l'eseguibile di Xboard dopo aver configurato il software con il relativo flag;

```
configure --enable-zippy
make
```

per Winboard invece non bisogna adottare nessun accorgimento particolare, dal momento che il codice di Zippy è già integrato nell'eseguibile `WinBoard.exe`.

Per usare il programma è sufficiente aggiungere il flag `–zp`, oppure `–zippyPlay True`. Vediamo un esempio di una riga di comando per aprire una connessione su FICS[4] utilizzando Zippy.

```
xboard –ics –icshost freechess.org –zp
```

Vediamo un paio di opzioni utili che possono servire qualora si utilizzi Zippy su ICS.

- `–zippyAcceptOnly` *`string`* serve per accettare automaticamente sfide soltanto da un ben specifico utente,
- `–zippyVariants` *`list`* è utilizzato per accettare partite di varianti di scacchi solo se la variante in questione è specificata nella lista.

6.6 CMail

[4] È un Internet Chess Server. Vedi capitolo 7.

CMail è un programma distribuito con Xboard che permette di giocare partite di scacchi via mail con supporto grafico. CMail non è portabile su Windows, quindi non funziona con Winboard.

Per lanciare il programma è sufficiente digitare il comando `cmail`; sarà quindi possibile dare un nome alla partita oppure lasciarla indefinita. Quindi si può specificare un indirizzo di posta oppure il nome o lo pseudonimo di uno dei nostri avversari, se li abbiamo memorizzati precedentemente.

CMail provvede a lanciare Xboard, l'utente effettua la mossa e impartisce il comando *Mail Move* dal menu *File*. CMail spedisce la mail all'avversario.

Quando arriva l'e-mail di risposta del nostro avversario è sufficiente, se il programma di gestione della posta lo permette, metterla in pipe (`|`) con `cmail`, oppure salvare il messaggio nel file *`filename`* e quindi digitare `cmail < `*`filename`*. Il programma ci farà vedere la mossa aprendo automaticamente Xboard.

Al momento di eseguire il programma è possibile aggiungere molte opzioni; si veda la documentazione per un elenco esaustivo.

7 Internet Chess Server

7.1 Cosa sono

Gli Internet Chess Server sono dei server ai quali un utente può connettersi e giocare contro altri esseri umani oppure contro programmi per scacchi. È possibile anche soltanto chattare con altri appassionati o visionare partite giocati da altri.

La maggior parte degli Internet Chess Server, oggi, sono diretti discendenti dell'ICS originale, scritto da Micheal Moore e Richard Nash nel 1992. Il codice degli Internet Chess Server, col passare degli anni, è stato riscritto ed ottimizzato, ma la compatibilità di questi server con ICS è rimasta. Per questo ancora oggi si usa l'espressione "ICS-compatibile" e si tende a sviluppare i vari programmi (per esempio Winboard/Xboard o Zippy) seguendo il protocollo ICS.

7.2 Il login

Innanzitutto prima di giocare bisogna loggarsi al server. Questo può essere fatto manualmente: se per esempio ci si vuole connettere via telnet è sufficiente impartire un comando del tipo

```
telnet hostname portnumber
```

Se non si dispone di alcuna interfaccia grafica bisognerà interagire da console, inserendo i comandi e i messaggi al prompt e ricevendo l'output sempre a video.

La connessione può essere altrimenti effettuata attraverso un'opportuna interfaccia. Molti ICS, infatti, mettono a disposizione un software (di solito gratuito) che offre tutte le funzionalità di cui si può aver bisogno per interagire con quel preciso server, per esempio:

- un'interfaccia grafica per giocare le partite,
- dei pulsanti appositi per effettuare comodamente la connessione al server,
- una finestra per mandare, ricevere messaggi, chattare eccetera,
- una serie di pulsanti e menu a tendina che corrispondono ai comandi propri del server,
- la possibilità, attraverso gradevoli diagrammi o grafici, di scegliere l'avversario opportuno,
- la possibilità di salvare e caricare file contenenti partite.

Una volta connessi bisogna effettuare il login, cioè inserire il nome dell'utente e la password, per dare la possibilità al server di identificarci. Se stiamo entrando in un ICS per la prima volta e non c'è modo di registrarsi prima, dobbiamo innanzitutto entrare come *guest*, e quindi chiedere di essere registrati.

Registrarsi ad un ICS (di solito) costa. Si può comunque entrare come guest gratuitamente. La differenza tra un utente ospite e un utente registrato è che a quest'ultimo è permesso usufruire di molti più servizi che al primo. Per la maggior parte dei server, comunque, è possibile giocare partite senza avere il bisogno di registrarsi.

7.3 Quali servizi offrono

Nel capitolo successivo vedremo in dettaglio le caratteristiche di uno di questi server; in questo paragrafo, quindi, diamo soltanto una descrizione generale dei servizi offerti dagli Internet Chess Server, per poi tornarci più approfonditamente nel capitolo 8.

Vediamo quindi cosa è effettivamente possibile fare una volta che ci si è connessi ad un ICS. Non tutti gli ICS offrono tutti i servizi che andiamo ad elencare; la lista

seguente, comunque, rappresenta un buon campione per capire le funzionalità messe a disposizione.

- accettare o rifiutare una sfida da parte di un utente,
- proporre una sfida ad un giocatore,
- cercare un giocatore da sfidare secondo determinati parametri,
- dialogare con il giocatore che si sta sfidando,
- osservare partite giocate da altri,
- essere informati automaticamente su sfide importanti che stanno per aver luogo,
- aver la possibilità di seguire un particolare giocatore, esaminando le sue partite non appena inizia a giocare,
- memorizzare e riesaminare partite,
- entrare in canali, unirsi a chatroom,
- ascoltare gli altri utenti ed eventualmente lasciare un messaggio sul canale.

7.4 La funzione timestamp/timeseal

Spesso un giocatore decide di giocare partite con limiti di tempo, un classico esempio sono gli scacchi tradizionali con un tempo di cinque minuti più un incremento di dodici secondi per ogni mossa. Il giocatore che non rispetta questi limiti perde la partita, da ciò si capisce che in questi casi il tempo è un parametro vitale.

Quando si gioca su un server però, i ritardi di rete possono causare dei disturbi nel conteggio del tempo; in assenza di adeguati strumenti il ritardo di rete rischia di venire conteggiato come tempo di thinking del giocatore. Questo è chiaramente ingiusto.

La funzione timestamp (su ICC) o timeseal (su FICS[1]) serve proprio a prevenire questo inconveniente: essa memorizza il thinking time del giocatore, in modo tale da non conteggiare anche il delay della trasmissione nel computo totale del tempo.

Per utilizzare le due funzioni con l'interfaccia Xboard è sufficiente usare uno dei due comandi

```
xboard -ics -icshost 204.178.125.65 -icshelper timestamp
```

[1] ICC (Internet Chess Club) e FICS (Free Internet Chess Server) sono due tra i più famosi Internet Chess Server, vedi paragrafo successivo.

```
xboard -ics -icshost 164.58.253.13 -icshelper timeseal
```

Il flag `–icshelper` informa il programma principale (Xboard) che c'è un programma esterno aggiuntivo (timestamp/timeseal) che comunica con il server.

Con Winboard, invece, per lanciare la funzione non bisogna fare nulla, dal momento che Winboard lancia automaticamente timestamp o timeseal, a seconda del server a cui abbiamo scelto di connetterci.

Tuttavia qualora un utente si trovasse a dover inserire il comando manualmente, le opzioni da aggiungere sono analoghe a quelle viste per Xboard, cioè `/icshelper timestamp` oppure `/icshelper timeseal`.

Mentre le due funzioni non sono comprese nella distribuzione di Xboard e vanno quindi scaricate dai siti ftp dei server, la distribuzione di Winboard comprende sia il programma `timstamp.exe` che `timeseal.exe`.

7.5 Alcuni Internet Chess Server

Π *Internet Chess Club – www.chessclub.com*. Lo vedremo nel prossimo capitolo.

Π *FICS (Free Internet Chess Server) – www.freechess.org*

Π *Chess.net – www.chess.net*

Π *The Internet Gaming Zone – www.zone.com*

Π *Playsite – www.playsite.com*

Π *Yahoo! Games: Chess - http://games.yahoo.com/games/login?game=Chess*

Π *Caissa's Web – www.caissa.com*

Π *Internet Chess Tournaments - http://chess.webest.com/*

Π *The Friendly Chessboard - http://www.goloops.com/chess/*

Π *Astro Chess Club - http://www.nhastro1.org/*

Π *Chess Line - http://151.17.13.5/chessline/homepage.asp*

Π *Ichess - http://www.ichess.com/*

Π *GCS (Global Chess Server) - http://global.chessparlor.com/*

Π *Chess-Square - http://www.chess-square.com/*

Π *International Champion Chess Network - http://www.icchess.net/*

Π *FIDE Game Zone - http://game.fide.com/*

Π *Pogo.com - http://www.pogo.com/*

Π *VOG Chess Club - http://www.vog.ru/chess*

8 ICC – Internet Chess Club

8.1 Introduzione

In questo capitolo vedremo in dettaglio le caratteristiche di ICC, Internet Chess Club (*www.chessclub.com*), uno dei più famosi e frequentati server scacchistici.

Vedremo cosa bisogna fare per accedervi, quali sono i servizi che mette a disposizione, i comandi più importanti; daremo inoltre uno sguardo al software *BlitzIn*, proprio del server, e anche ad alcuni aspetti un po' più tecnici, come il protocollo di scambio dei dati tra il server e un computer connesso alla rete.

8.2 Storia di ICC

Internet Chess Server è il più antico tra i server scacchistici che si trovano in rete. È a tutt'oggi la più vasta organizzazione di giocatori di scacchi: non è raro che gli utenti collegati in un dato istante siano più di duemila.

Internet Chess Server è nato negli anni ottanta, in seguito al riconoscimento, da parte di una piccola comunità non ufficiale di scacchisti, del vasto potenziale di Internet relativamente al gioco degli scacchi. Nel 1992 un team di programmatori si propose di riscrivere il codice del server, in maniera tale da renderlo elegante, più efficiente e per permettere a chiunque di utilizzare i servizi di ICC. Questo compito venne portato a termine mantenendo la compatibilità con la versione originale.

ICC è stato il primo server e anche dopo la nascita di altri, per molti anni è stato comunque il primo e unico a fornire certi servizi. Con gli anni anche i server

concorrenti hanno adeguato la loro offerta. Per esempio, Internet Chess Server è stato il primo ad ospitare un'esibizione di un campione del mondo (Garry Kasparov, 1995).

ICC è amministrato da un gruppo di appassionati che si prefiggono di mantenere un alto livello di privacy per gli utenti, di sportività tra i giocatori e di salvaguardare l'integrità del sistema di rating.

8.3 Il software BlitzIn

Dalla home page del sito è possibile raggiungere con facilità l'area download, dove è possibile scaricare gratuitamente il software BlitzIn.

Si tratta di un'interfaccia grafica per il sistema operativo Windows, che mette a disposizione molte funzioni, facendo sì che l'utente possa connettersi al server, richiedere la maggior parte dei servizi e impartire la maggior parte dei comandi propri del server, per mezzo di pulsanti e menu a tendina.

Vedremo le caratteristiche principali del software nel corso del capitolo, via via che illustreremo i servizi di cui possiamo usufruire.

La versione di BlitzIn esaminata è la 2.31.

8.4 Altri tipi di interfaccia

Dal momento che BlitzIn è un'interfaccia che funziona soltanto col sistema operativo Windows, l'utente che usa una piattaforma differente deve utilizzare un'altra interfaccia. ICC mette a disposizione un'area del sito in cui è possibile scegliere l'interfaccia a seconda del sistema operativo che si usa.

Innanzitutto, oltre a BlitzIn, sono consigliate altre interfacce per Windows, tutte ICC compatibili:

- Icarus (*http://www.randomly.com/icarus/*),
- Chess Assistant 6 (*http://www.chessclub.com/store/chessassistant/*),
- Winboard (*http://www.tim-mann.org/chess.html*).

Per piattaforma Unix/Linux è consigliata Xboard. Sono poi considerate alcune altre piattaforme, tra cui MacOS.

Oltre ad interfacce grafiche vere e proprie è possibile scaricare interfacce Java, un po' meno raffinate e con qualche funzionalità in meno, ma comunque valide.

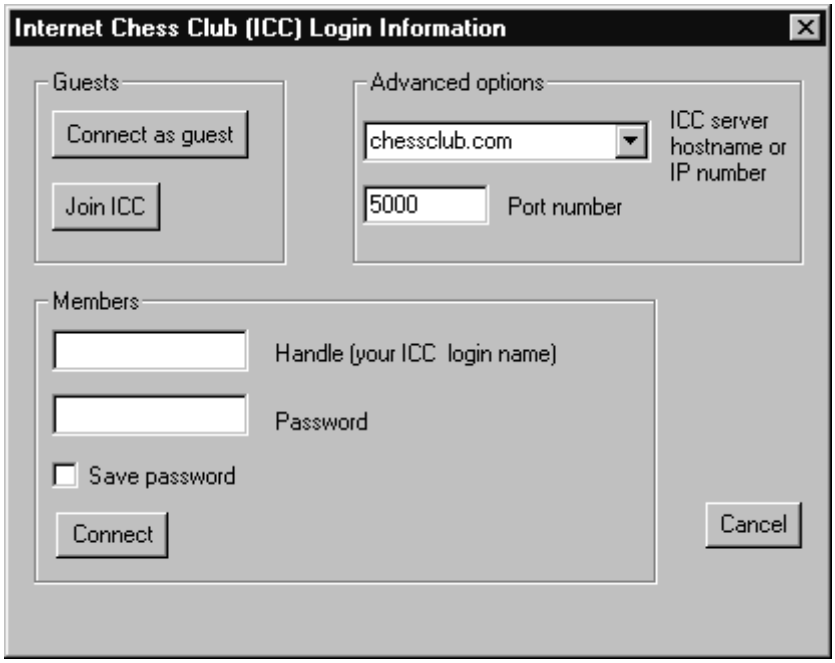
- SimpleChess (*http://queen.chessclub.com/sji/index.html*),
- EasyChess (*http://www.chessclub.com/easychess/*),
- CoffeeHouse (*http://www.chessclub.com/CoffeeHouse.html*),
- Jin (*http://www.jinchess.com/*).

8.5 La connessione

Come per gli altri server anche per ICC è possibile entrare come *guest* oppure registrarsi e diventare membro. Naturalmente la registrazione comporta una spesa.

La connessione può essere effettuata nei modi che abbiamo già visto: senza usare interfacce grafiche (via telnet, per esempio) oppure attraverso un'interfaccia. Se stiamo usando BlitzIn non occorre naturalmente specificare a quale server vogliamo connetterci, dal momento che tale software è proprio di ICC e quindi provvede automaticamente ad aprire una connessione con quel server. Vediamo come si fa.

Cliccando sul menu *File* e scegliendo l'opzione *Nuova Connessione* ci appare la finestra



dalla quale è possibile scegliere se entrare come guest, registrarsi oppure inserire nome utente e password se si è già membri.

8.6 Giocare una partita su ICC

Una volta connessi ci appare la console,

dove possiamo leggere messaggi indirizzati a tutti, messaggi personali, proposte di gioco, e dove possiamo a nostra volta scrivere dei messaggi.

8.6.1 Cercare uno sfidante

Per cercare il giocatore contro cui giocare una partita ci sono molti modi.

Se per esempio vogliamo sfidare un giocatore ben preciso, secondo parametri ben definiti possiamo farlo col comando

```
match user time inc
```

time e *inc* sono i parametri che regolano i vincoli di tempo della gara. Lo stesso comando si può impartire da uno dei menu di BlitzIn, lo vedremo in un paragrafo successivo.

Per accettare o rifiutare è sufficiente digitare

```
accept user     oppure     decline user
```

Per cercare un qualsiasi giocatore che acconsenta a giocare una partita secondo determinati parametri si impartisce un comando del tipo

```
seek 3 10
```

La richiesta apparirà a video con la forma

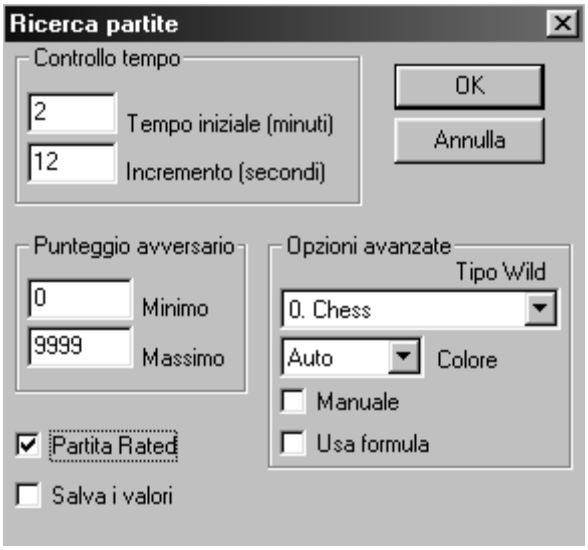
```
myname seeking blitz 3 10 rated ("play 8" to respond)
```

all'utente interessato sarà sufficiente digitare

```
play 8
```

per cogliere la sfida.

Col software BlitzIn tutto è molto più facile e comodo. Dal menu *Game*, scegliendo *Seek a Game…*, comparirà la seguente finestra di dialogo:

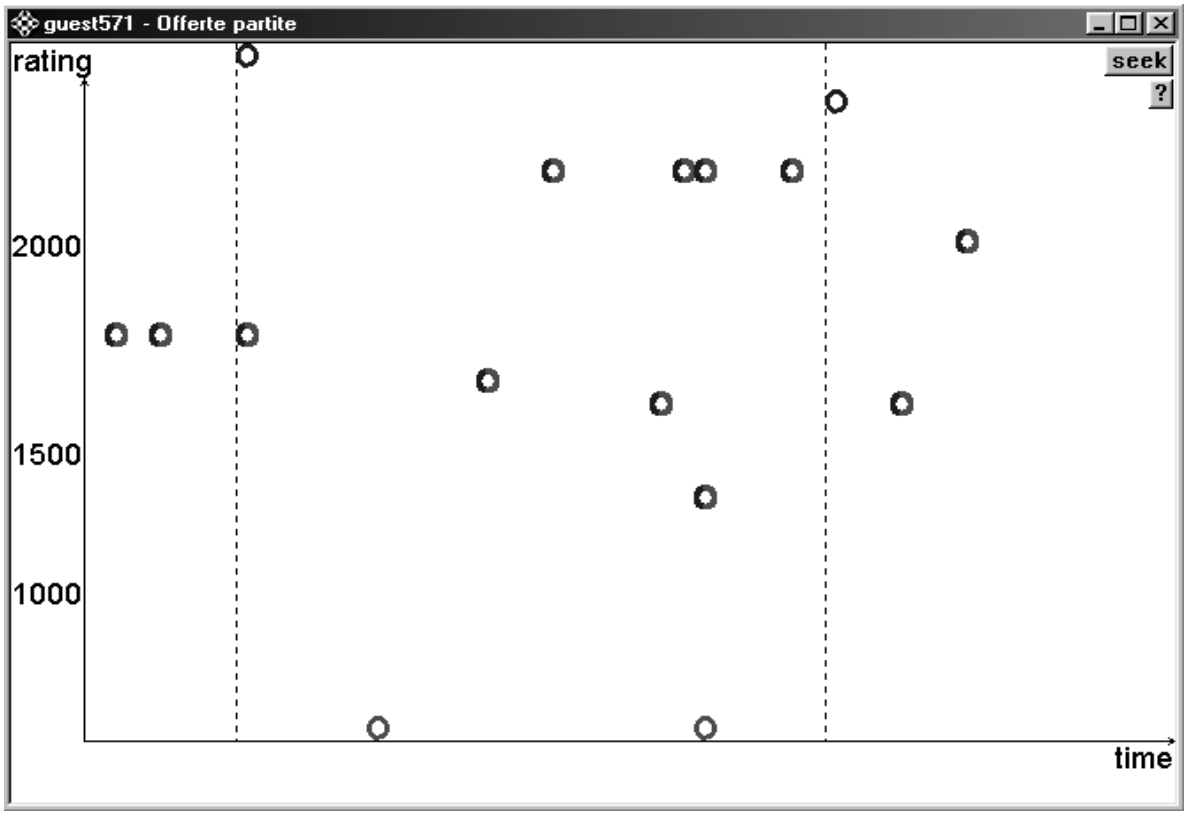


da cui si possono specificare i parametri di tempo desiderati. Si può inoltre precisare se la partita sia *rated* o *unrated* (che influenzi o meno il punteggio

dell'avversario), indicando il range di punteggio desiderato, oltre che decidere se giocare a scacchi tradizionali o ad una variante.

Una volta premuto *ok*, appena un giocatore accetta di confrontarsi con noi, la partita inizia automaticamente.

Un altro modo per cercare uno sfidante è quello di usare il *Seek Graph*, opzione presente nel menu *Window*.



Si tratta di un grafico bidimensionale che ha per ascissa i parametri di tempo della partita e per ordinata il rating dello sfidante. Gli anelli all'interno del grafico rappresentano gli attuali potenziali avversari. Più un anello è spostato verso destra più i parametri di tempo sono alti; più è spostato verso l'alto più lo sfidante è forte.

A seconda del tipo di anello, inoltre, capiamo se il giocatore è una macchina (anello colore blu) o un essere umano (verde) se gioca ad una variante degli scacchi (colore rosso) rispetto agli scacchi tradizionali, se la partita è rated (anello pieno) o unrated (vuoto). Gli anelli per metà colorati di blu e per metà di rosso rappresentano programmi che giocano a qualche variante.

Passando sopra col mouse in basso compare una scritta che ci dà qualche informazione in più sul giocatore, per esempio:

RoBoT (C) 1400 seeks wild(20) 10 0 unrated black
Guest458 (U) seeks Standard 45 12 unrated

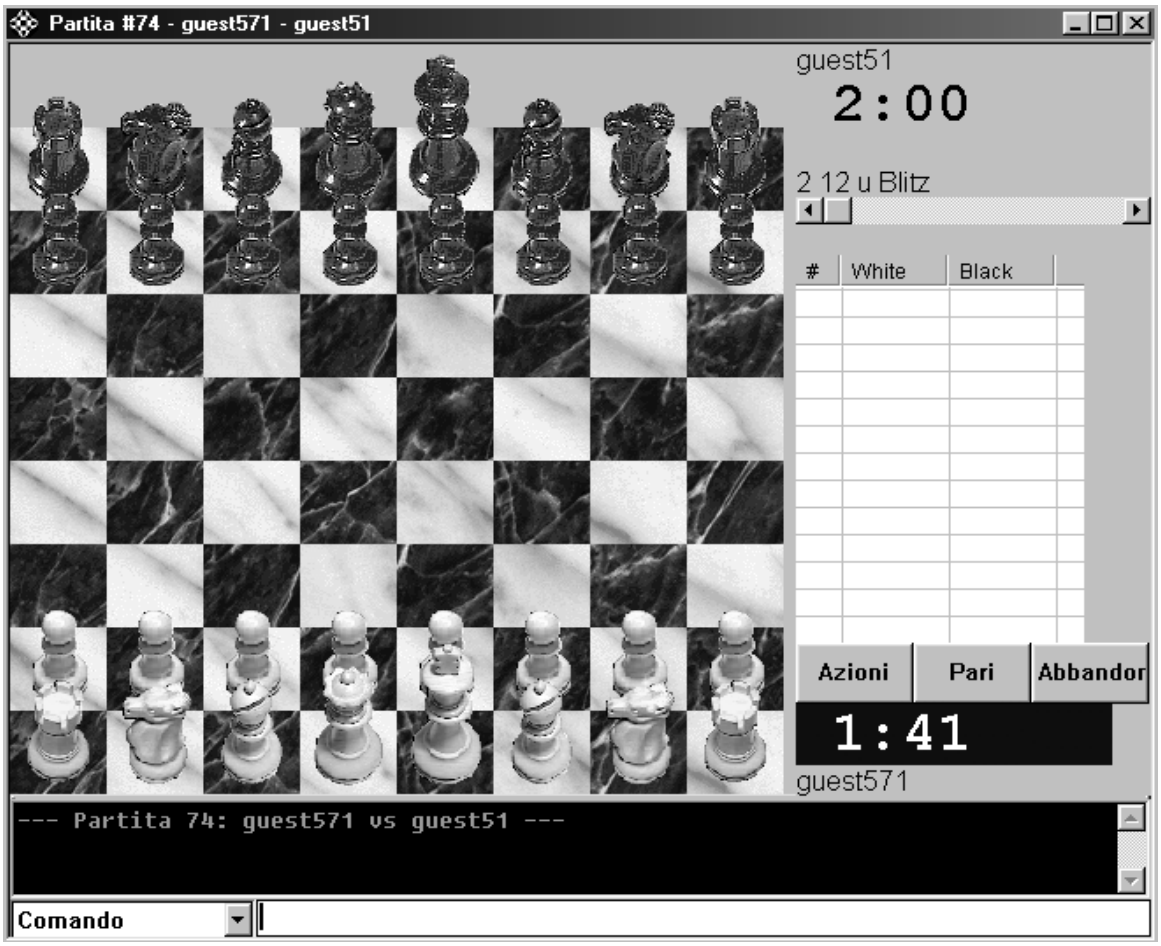
La prima ci informa che un programma di nome *RoBoT*, di forza 1400, cerca uno sfidante per giocare alla variante numero 20, con parametri di tempo di 10 minuti per ogni mossa senza incremento, senza rating e con pezzi neri.

La seconda rappresenta un essere umano, entrato come guest, che cerca uno sfidante per scacchi tradizionali, senza rating, con 45 minuti a disposizione per ogni mossa, più eventualmente 12 minuti di incremento.

8.6.2 La partita

Una volta scelto l'avversario la partita inizia. Se stiamo usando BlitzIn ci appare la scacchiera. Sulla barra dei comandi di BlitzIn ci sono moltissime opzioni per cambiare le caratteristiche della scacchiera: colore dei pezzi, tipo di scacchiera, visione bidimensionale o tridimensionale, solo per citarne alcuni. Nella pagina seguente ci sono due immagini raffiguranti due possibili visioni della scacchiera: la prima immagine mostra la scacchiera di default.


Partita #163 - guest571 - guest689
guest689
2:28
2 12 u Blitz
White Black
1 Nc3 e5
2 e4 Bb4
3 a3 Bxc3
4 dxc3 Nf6
5 Qe2 h6
6 Nf3 d6
7 Qb5+
Azioni
Pari
Abbandon
2:35
guest571
--- Partita 163: guest571 vs guest689 ---
Comando


Partita #74 - guest571 - guest51
guest51
2:00
2 12 u Blitz
White Black
Azioni
Pari
Abbandor
1:41
guest571
--- Partita 74: guest571 vs guest51 ---
Comando

La mossa viene effettuata semplicemente spostando il pezzo con il mouse. Qualora invece non disponessimo di un'interfaccia grafica dobbiamo inserire le mosse manualmente, da console. ICC usa la notazione SAN; nel caso una mossa sia ambigua o illegale (per esempio se muovendo il proprio re viene lasciato sotto scacco) il server comunica al giocatore un messaggio d'errore.

Una partita può terminare, oltre che in caso di vittoria di uno dei due giocatori, a seguito dell'esecuzione dei seguenti comandi:

- `abort`: la partita viene interrotta,
- `adjourn`: la partita viene interrotta per poi eventualmente poter essere ripresa,
- `draw`: richiesta di patta,
- `flag`: se c'è un solo giocatore che ha ecceduto i vincoli di tempo la vittoria va all'avversario,
- `resign`: vince l'avversario per abbandono.

Naturalmente questi, come tutti gli altri comandi, possono essere inseriti manualmente o usando BlitzIn o un'altra interfaccia grafica.

8.7 Comunicare su ICC

8.7.1 Messaggi personali: il comando `tell`

Per parlare con un utente particolare si utilizza il seguente comando

```
tell user message
```

per ripetere il comando, indirizzando il messaggio allo stesso utente, si può usare il comando

```
. message
```

Questo è molto comodo se si sta facendo una conversazione piuttosto lunga. Vediamo un esempio dei due comandi precedenti:

```
tell paul how are you today?
(told paul)
paul tells you: fine, and yourself?
. pretty good.
(told paul)
```

8.7.2 Canali

Per parlare con più utenti ICC contemporaneamente è possibile accedere a uno dei 300 canali presenti. Ad ogni canale è associato un topic particolare; gli utenti non registrati sono ammessi soltanto nei canali 1 e 50, per chiedere informazioni su problemi di carattere generale. Digitando `help channel` è possibile avere informazioni su tutti i comandi che riguardano i canali, nonché visualizzare la lista completa di tutti i canali con i relativi topic.

Per entrare in un canale si digita il comando

`+channel` *`number`*

per uscirne

`–channel` *`number`*

Per parlare in un determinato canale si usa

`tell` *`number message`*

per ripetere il comando con un messaggio diverso invece si scrive

`,` *`message`*

Per visualizzare tutti gli utenti presenti nel canale in un determinato istante si usa il comando

`inchannel` *`number`*

Il canale 1 è un canale particolare, detto *help channel*. Viene usato da chiunque abbia un dubbio su qualsiasi questione riguardante il server. La domanda viene

valutata da un gruppo di amministratori (indicati col simbolo `(*)`), che tentano di rispondere al quesito in breve tempo. Vediamo un esempio;

```
    tell 1 I’m a programmer. Where can I get information about
writing client programs for ICC?
    (submitted to 29 admins and helpers)
    Zek(*)(1): Guest571 wrote: I’m a programmer. Where can I get
information about writing client programs for ICC?
    Zek(*)(1): type “help programmers”
```

8.7.3 Messaggi

È possibile mandare messaggi anche ad un utente che momentaneamente non è on-line. Per esempio:

```
message SylviaXX Hello
Added the following message to SylviaXX:
--Marko (11:53 29-Aug-02 EDT): Hello
```

Non appena il destinatario effettuerà il login potrà leggere il messaggio. Digitando `messages` è possibile infatti vedere tutti i messaggi che ci sono arrivati.

8.7.4 Comunicare durante una partita

Per comunicare durante il gioco di solito si usano i tre comandi `say`, `kibitz` e `whisper`. Tutti e tre seguono la sintassi

```
command message
```

Il comando `say` serve per mandare un messaggio all’avversario contro il quale si sta giocando. Il comando `kibitz`, invece, serve per inviare dei commenti sulla gara sia all’avversario che agli eventuali osservatori. Infine il comando `whisper` è indirizzato ai soli osservatori e non all’avversario.

8.8 Informazioni sugli utenti

Per gli utenti registrati vengono memorizzate delle informazioni che possono essere lette da chi ne ha bisogno. Questi sono i comandi più utili:

- `finger`: visualizza il finger file di un utente. Tale file contiene informazioni di carattere generale (nome, indirizzo e-mail eccetera). Se l'utente è una macchina nel finger file deve comparire il nome del programma, la versione e il calcolatore su cui il programma gira.
- `history`: visualizza l'elenco delle partite giocate da un utente.
- `who`: visualizza la lista di tutti gli utenti attualmente loggati su ICC.

Le informazioni relative ad un utente sono memorizzate attraverso un database, che permette tra le altre cose di mantenere aggiornato il rating di ogni giocatore.

Quando si visualizza la lista parziale o totale degli utenti loggati è possibile, attraverso opportuni simboli, capire se il giocatore è disponibile per una partita, se invece sta giocando o se sta esaminando una partita.

Dopo il rating e il nome del giocatore è possibile trovare un simbolo che specifica la natura del giocatore;

GM	Grandmaster
IM	International Master
FM	FIDE Master
WGM	Woman Grandmaster
WIM	Woman International Master
DM	ICC Display Master
TD	Tournament Director
C	A computer
*	An ICC administrator
H	An ICC helper

8.9 Osservare ed esaminare le partite

ICC offre la possibilità di osservare partite giocate da altri. Spesso agli utenti registrati viene comunicato ogni qual volta sia in programma una partita particolarmente importate, per esempio tra due Maestri.

Con il comando `games` è possibile ottenere una lista di tutte le partite che si stanno giocando. Ogni partita è identificata da un *game number*. Con il comando

```
observe game_number
```

si inizia ad osservare la partita *`game_number`*.

Con il comando

```
observe username
```

è possibile osservare la partita che sta giocando l'utente *username.* Il comando `unobserve` esce dalla modalità *observe*.

Un altro comando interessante è `follow`.

```
follow username
```

segue le azioni dell'utente *`username`*, osservando la partita, se l'utente sta giocando, o in caso contrario attivando la funzione *observe* automaticamente non appena *`username`* inizia una nuova gara. Il comando viene disattivato con `unfollow`.

Osservare una partita è un concetto diverso dall'esaminare. Mentre l'osservazione implica che due giocatori stiano giocando una gara in quell'istante, esaminare una partita significa analizzare una gara memorizzata nel database *history*.

Abbiamo visto che col comando

```
history username
```

possiamo accedere alla lista di tutte le partite giocate dall'utente *`username`*. Ora, se desideriamo esaminare la partita numero 12 nella lista di quell'utente si usa il comando

```
examine username 12
```

Tale comando viene terminato digitando `unexamine`.

Mentre si sta esaminando una partita è possibile muoversi attraverso le mosse con i comandi `backward` e `forward`, oppure tentare qualche variante inserendo la mossa che preferiamo per poi tornare indietro col comando `revert` e proseguire l'analisi.

Per un elenco completo dei comandi di ICC digitare `help` mentre si è connessi al server, oppure leggere il documento *Manual-help* accessibile dal sito.

8.10 Il formato dei dati di output

Recentemente il formato dei dati in output, cioè dal server al nostro programma, è stato rivisto. In conseguenza di questo fatto si è distinto il vecchio formato, detto livello 1, da quello nuovo, il livello 2. I due tipi di formato coesistono, rendendo possibile ai nuovi programmi di fare cose che prima non potevano e, contemporaneamente, garantendo la compatibilità dei vecchi programmi.

Questo è il comportamento del server:

- riceve dal client una sequenza di stringhe, ognuna delle quali è un comando,
- il server esegue i comandi secondo l'ordine in cui arrivano; non esegue un comando se prima non ha terminato il comando precedente,
- ogni comando fa sì che dei dati vengano spediti a uno o più utenti; al client che ha inoltrato il comando viene spedito un responso, chiamato *unit* o anche *datagram.*

8.10.1 Livello 1

L'output di livello 1 viene formattato in modo che le unit risultino chiaramente delimitate. Ogni stringa contiene il *command number* e l'utente che ha generato il comando. Lo scopo del livello 1 è quindi, semplicemente, quello di raggruppare gli output di ICC in unità discrete, corredate del nome del client dal quale il comando proviene.

Il livello 1 rende possibile per ogni client capire chiaramente quando l'output di un comando termina, che tipo di comando era e qual è stato il client che lo ha inoltrato.

Un datagram di livello 1 ha la forma

```
^Y[  command_number  player  "command_output"  ^Y]
```

quindi, quando il client legge la sequenza `^Y]` sa che l'output del comando è terminato. I datagram possono essere anche annidati; vediamo un esempio (l'asterisco indica che il datagram viene mandato al client che ha inoltrato il comando):

```
^Y[328 fishbait
fishbait spoofs you: "partner fishbait"
^Y[199 *
Sending a 'tell' message to inform your chosen partner...
^Y[101 *
Not sent -- fishbait does not get messages from unregistered
   players.
^Y]^Y]^Y]
```

I command number sono dei codici associati a tutti i comandi che il server è in grado di comprendere, vediamo qualche esempio :

```
#define CN_TELL                         101
#define CN_MATCH                        121
#define CN_INCHANNEL                    158
#define CN_RESIGN                       208
```

8.10.2 Livello 2

Lo scopo del livello 2 è mettere a disposizioni delle funzioni che non erano contemplate nel livello 1, come ad esempio

- mantenere una lista di tutti gli utenti loggati, compreso il loro rating,
- mantenere una lista di tutte le partite in corso,
- mantenere una lista di tutti gli osservatori della partita che si sta giocando,
- mantenere una lista di mosse della partita che si sta giocando od osservando.

Un datagram di livello 2 ha la seguente forma:

```
^Y( data_type_number arg1 arg2 arg3 … argn ^Y)
```

dove *data_type_number* è un numero che indica il tipo di azione da eseguire, per esempio:

- `1 DG_PLAYER_ARRIVED`: serve quando si vuole mantenere una lista di utenti connessi. Ogni volta che un nuovo utente effettua il login viene spedito al client un datagram con numero `DG_PLAYER_ARRIVED`, seguito dal nome del nuovo utente,
- `2 DG_PLAYER_LEFT`: inviato quando un giocatore si disconnette,
- `12 DG_GAME_STARTED`: è utile quando si ha la necessità di mantenere una lista di tutte le partite in corso. Appena una nuova partita ha inizio viene mandato al client un datagram `DG_GAME_STARTED` con tutte le possibili informazioni che riguardano il match. Questa la sua forma:

```
(gamenumber whitename blackname wild-number rating-type
rated  white-initial white-increment black-initial black-
increment played-game white-rating black-rating game-id
white-titles black-titles irregular-legality irregular-
semantics uses-plunkers fancy-timecontrol)
```

- `20 DG_PLAYERS_IN_MY_GAME`: usato per tenere traccia degli utenti che stanno osservando la nostra gara.

9 Sviluppo di un programma per il salvataggio automatico di file PGN da ICC

9.1 Introduzione a PGN-Saver

PGN-Saver è un programma realizzato dall'autore del presente elaborato. Si tratta di un agente automatico per il salvataggio di partite in formato PGN da Internet Chess Club.

Utilizzando i principi che governano il protocollo di output di ICC visti nel capitolo precedente (paragrafo 8.10) il programma si connette al server e scarica le partite dei giocatori scelti dall'utente. Le partite vengono salvate una per file.

I salvataggi vengono effettuati in base al tipo di gioco (scacchi tradizionali, varianti eccetera), anche questo specificato dall'utente.

Il programma è progettato per fungere da demone. È possibile infatti ordinare al programma di effettuare un certo numero di connessioni, intervallate da un certo lasso di tempo. In questo modo si può eseguire la connessione una volta, più volte o un numero infinito di volte.

Vengono salvate soltanto le partite nuove, cioè soltanto quelle partite che non sono state salvate in una precedente connessione.

Il formato dei file è PGN, leggibile da quasi tutti i software per scacchi. Dopo aver salvato il file è sufficiente quindi aprirlo con l'interfaccia (o database, o altro software) preferita, oppure semplicemente cliccarci sopra se è stato impostato un visualizzatore di default per file PGN.

Il programma è stato scritto in linguaggio C e sviluppato in ambiente Unix. Non è compatibile con altre piattaforme.

9.2 Compilare e lanciare il programma

Il codice del programma è contenuto nella directory `pgn-saver/source`. Per la compilazione porsi a livello della directory `pgn-saver` e digitare

```
make
```

Per lanciare il programma, sempre a livello della directory `pgn-saver`, impartire il comando

```
./pgns
```

I file in formato PGN vengono salvati nella directory `download`, creata automaticamente. La directory `download` e il file di log vengono creati nella stessa directory dove si trova l'eseguibile, cioè `pgn-saver`.

9.3 Comandi

Vediamo nel dettaglio tutti i servizi messi a disposizione dal software e, per ognuno di questi, i comandi che l'utente può impartire all'interprete.

9.3.1 Salvare le partite

Il comando principale del programma, quello usato per effettuare la connessione e salvare i file in formato PGN, è

```
start
```

Una volta inserito questo comando l'interprete va a leggere i nomi dei giocatori da un file. Perché questo avvenga correttamente l'utente deve prima aver creato un

file di nome `pgns.pls` nella directory `pgn-saver`, che contiene i nomi dei parametri da passare all'agente[1].

Il buffer in cui il file viene letto ha una dimensione massima di 1000 caratteri. La dimensione massima del nome di un giocatore è di 16 caratteri (tuttavia 15 dovrebbero essere sufficienti, considerando il limite imposto da ICC). Ciò significa che nel peggiore dei casi (cioè tutte le handle sono composte da 15 caratteri, più uno spazio o un newline per separare le handle) il numero massimo di handle specificabili è di 62. Se, come poi accade nella realtà, le handle dei giocatori hanno una lunghezza media di circa otto caratteri, il numero massimo di parametri sale a oltre 100. È possibile modificare questo valore intervenendo sulla dimensione del buffer `PLAYERS_BUFFER_SIZE`, nel file `pgn-saver/source/h/def.h`.

Se il file `pgns.pls` non esiste viene visualizzato un messaggio d'errore.

Oltre a leggere i parametri da file c'è un altro modo con cui è possibile specificare le handle dei giocatori, e cioè passando manualmente le handle al comando `start`.

Per fare questo occorre prima di tutto cambiare modalità di input, col comando

```
input keyboard
```

A questo punto il comando `start` assume la forma

```
start user1 user2 user3 ...
```

dove `user1 user2 user3 ...` sono i nomi (di account ICC) dei giocatori per i quali si vogliono salvare le partite. Per esempio il comando

```
start hazel killerqn Renato
```

salva le partite dei tre giocatori hazel, killerqn e Renato.

[1] Si veda il paragrafo successivo per la descrizione del formato del file `pgns.pls`.

È importante fare attenzione a come viene impartito tale comando. Innanzitutto, in modalità con input da tastiera, l'interprete non ammette che il comando non abbia parametri:

```
start
```

non viene accettato.

Gli spazi sono essenziali: fungono da delimitatori tra un parametro e l'altro; per esempio il comando

```
start  hazel killerqn Renato
```

con due spazi tra il comando e il primo parametro, potrebbe causare problemi sia nel conteggio dei parametri che nelle handle dei giocatori, che potrebbero venire disturbate dallo spazio in più.

La dimensione massima del vettore contenente le handle dei giocatori è sempre di 1000 caratteri. Il numero massimo di handle specificabili è quindi lo stesso del caso di input da file. È possibile modificare questo valore intervenendo sulla dimensione del vettore dei parametri `START_ARGS_SIZE`, nel file `pgn-saver/source/h/def.h`.

La modalità di input da file è l'opzione di default. Se comunque per qualche motivo l'utente ha scelto la modalità da tastiera e desidera ritornare a quella da file può farlo inserendo il comando

```
input file
```

Sono escluse dal salvataggio le partite interrotte (aborted).

9.3.2 Il formato del file di input

Il file `pgns.pls` deve seguire un formato ben preciso, per dare modo all'interprete di riconoscere le handle dei giocatori senza ambiguità.

- Le handle devono iniziare dal primo byte del file,
- le stringhe con i nomi degli utenti devono essere separate da uno e un solo carattere separatore (spazio o newline),
- dopo l'ultima handle è necessario aggiungere almeno due caratteri separatori (spazio o newline) affinché l'interprete possa stabilire dove termina la lista di parametri.

L'interprete riconosce i seguenti tipi di formati errati e non li accetta:

- file privo della coppia di separatori alla fine,
- file in cui le handle non iniziano dal primo byte,
- file formattato correttamente ma privo di handle,
- file vuoto.

Ecco tre esempi di file corretti:

```
'hazel killerqn Renato  '
'hazel killerqn\nRenato\n\n'
'hazel\nkillerqn\nRenato\n  \n\n\n'
```

danno inizio alla connessione che salva le partite di hazel, killerqn e Renato.

Il comando

```
checkformat
```

visualizza il formato del file `pgns.pls`, compresi spazi (`sp`) e newline (`nl`).

Qualora il file non venga accettato dall'interprete viene suggerito di invocare il comando `checkformat` per controllare che il formato sia rispettato.

9.3.3 Rimozione di handle incorrette

Con il comando

```
remove on
```

si ordina al programma di rimuovere le handle incorrette dal file `pgns.pls`.

Nel solo caso di input da file, qualora il programma tenti di salvare le partite di un giocatore e si veda rispondere dal server che il nome dell'account non è corretto il programma provvede a rimuovere la handle relativa dal file `pgns.pls`. Questo perché, col trascorrere del tempo, è possibile che un account cessi di esistere.

Il file `pgns.pls` viene aggiornato e ri-formattato automaticamente (la formattazione scelta è quella con il newline come carattere separatore, quindi può accadere che l'utente editi il file usando come carattere separatore lo spazio e successivamente se lo trovi formattato con il newline).

Al termine del programma, nel caso di rimozioni di handle, viene visualizzato un messaggio di warning che mostra il numero di handle rimosse.

Con il comando

```
remove off
```

si disattiva la funzione di rimozione di handle incorrette.

Il valore di default per questa opzione è `ON`.

9.3.4 Tipi di partite da salvare

L'utente può specificare il tipo di gara che deve essere salvato. Esistono diversi tipi di gara; ecco alcuni simboli usati da ICC:

`b` = blitz,
`s` = standard,
`B` = bullet,
`d` = bughouse.

Oltre a questi è possibile specificare alcune durate di partita attraverso un simbolo, per esempio

`o` = partite da un minuto,
`f` = partite da cinque minuti.

Questi simboli occupano le prime due posizioni del campo *Type* dell'output del server (le posizioni totali sono tre). Le posizioni necessarie sono due affinché sia possibile specificare il numero della variante per scacchi. Ogni variante è infatti identificata da un numero a una o due cifre: numeri da una cifra (0-9) per le prime dieci varianti e a due cifre per le successive.

La terza posizione serve per specificare se la partita è *rated* (influenza il punteggio dei giocatori) oppure *unrated*.

`r` = rated,
`u` = unrated.

Vediamo qualche esempio di tipi che si possono trovare:

`' or'`: partita da un minuto rated,
`' sr'`: scacchi tradizionali rated,
`' bu'`: blitz unrated,
`'16r'`: variante 16 (kriegspiel) rated.

Il tipo di partita da scaricare può essere specificato dall'utente attraverso il comando

```
type
```

Una volta inserito tale comando l'interprete chiede all'utente di inserire il tipo voluto. Non è necessario inserire spazi. Se, per esempio, vogliamo scaricare partite di blitz rated è sufficiente inserire

```
'br'
```

e non

```
' br'
```

anche se in effetti nel campo *Type* dell'output del comando *history* leggiamo `' br'`.

Nello specificare il tipo l'utente può inserire anche il carattere jolly (`*`). Il carattere jolly fa match con tutti i caratteri: in questo modo è possibile salvare più di un tipo di gara con un'unica sessione. Vediamo qualche esempio:

`'b*'` : salva tutte le partite di blitz, sia rated che unrated,
`'**r'`: salva tutte le partite rated, di qualsiasi tipo,
`'***'`: salva tutte le partite indipendentemente dal tipo.

Bisogna fare attenzione ad una cosa: il carattere jolly fa match con un solo carattere. Questo vuol dire che `*` fa match con lo spazio, ma uno spazio e un carattere non fanno match con `*`. Per esempio

`'*u'` : salva le partite unrated, ma non salva per esempio le partite `'10u'`, perché il 10 è composto da due caratteri.

Due caratteri jolly invece fanno match con uno spazio e un carattere. Quindi, per esempio

`'**u'`: salva tutte le partite unrated, sia del tipo `'bu'` che del tipo `'10u'`.

L'utente non deve necessariamente specificare il tipo ogni volta. Il valore di default è

`'16*'`: salva tutte le partite della variante 16 (kriegspiel), sia rated che unrated,

ma può essere modificato cambiando il valore del vettore `type`, nel file `pgn-saver/source/c/main.c`.

9.3.5 Intervallo tra connessioni

Quando il programma è impostato per effettuare più connessioni consecutive è possibile impostare l'intervallo tra una connessione e l'altra. Tra due connessioni diverse il programma non fa nulla, ma attende per un certo intervallo di tempo, nella speranza che alla connessione successiva siano state giocate altre partite (ricordiamo che il programma salva solo le partite nuove, cioè non salvate precedentemente).

Il comando usato per questo scopo è

```
interval
```

lanciato il quale l'interprete dei comandi chiede all'utente quale sia l'intervallo desiderato. Il valore rappresenta il numero di minuti che il programma deve attendere tra due connessioni. Tale valore deve essere intero e viene rifiutato se è minore di zero. È accettabile anche il valore zero: in questo caso il programma effettua più connessioni consecutive senza attendere tra una e l'altra.

Il valore di default (modificabile nel file `pgn-saver/source/h/def.h`) è di due ore (120 minuti).

9.3.6 Numero di connessioni

È possibile impostare il numero di connessioni che il programma effettua prima di terminare. Il comando è

```
loop
```

Dopo averlo inserito l'interprete di comandi chiede all'utente di inserire il numero di connessioni. Tale valore deve essere un intero e non vengono accettati valori minori di uno: ciò vuol dire che almeno una connessione va fatta.

È anche possibile ordinare al programma di effettuare infinite connessioni; ciò può rivelarsi utile quando si vuole utilizzare il programma come un demone sempre in funzione (magari eseguendolo in background). Il comando da usare in questo caso è

```
loop infinite
```

Il valore di default (modificabile dal file `pgn-saver/source/h/def.h`) per questa opzione è proprio il numero di connessioni infinito.

9.3.7 Tentativi di connessione

Col comando

```
ntry
```

si specifica il numero massimo di tentativi per la connessione. Questo è utile affinché il programma possa uscire quando la connessione non è proprio possibile, altrimenti l'utente rischia di non accorgersi che c'è qualche problema con la rete.

Il programma quindi tenta di connettersi. Se fallisce ritenta e se la connessione ha successo prima che il numero di tentativi si esaurisca il programma prosegue. Se invece tutti i tentativi a disposizione fanno registrare un fallimento il programma esce.

Il valore deve essere un intero non minore di uno.

Il valore di default (modificabile dal file `pgn-saver/source/h/def.h`) è cinque.

9.3.8 File di log

Il cosiddetto *log mode* è una modalità che se attivata salva in un file di log i risultati delle principali azioni compiute dal programma, nonché gli errori intervenuti durante l'esecuzione.

Il formato di una riga del file di log è

data ora: azione (o errore)

Vediamo un esempio di un file di log relativo a una sessione che ha effettuato tre connessioni a distanza di un'ora l'una dall'altra, salvando otto file la prima volta, due la seconda e nessun file la terza, senza rilevare errori:

```
Thu Sep 26 16:14:04 2002: --------------- NEW SESSION ---------------

Thu Sep 26 16:14:04 2002: Connected to server
Thu Sep 26 16:14:05 2002: Created file killerqn-PepperSpray.Sep.26.02.15:58.pgn
Thu Sep 26 16:14:06 2002: Created file killerqn-PepperSpray.Sep.25.02.15:50.pgn
Thu Sep 26 16:14:08 2002: Created file killerqn-Blacula.Sep.25.02.08:04.pgn
Thu Sep 26 16:14:08 2002: Created file killerqn-SilverStar.Sep.24.02.19:41.pgn
Thu Sep 26 16:14:09 2002: Created file killerqn-SilverStar.Sep.24.02.18:47.pgn
Thu Sep 26 16:14:10 2002: Created file killerqn-SilverStar.Sep.24.02.18:07.pgn
Thu Sep 26 16:14:11 2002: Created file killerqn-Onulet.Sep.24.02.15:45.pgn
Thu Sep 26 16:14:13 2002: Created file killerqn-Onulet.Sep.24.02.14:20.pgn
Thu Sep 26 16:14:15 2002: Disconnetted

Thu Sep 26 17:14:15 2002: Connected to server
Thu Sep 26 17:14:16 2002: Created file killerqn-PepperSpray.Sep.26.02.16:55.pgn
Thu Sep 26 17:14:17 2002: Created file killerqn-PepperSpray.Sep.26.02.16:17.pgn
Thu Sep 26 17:14:20 2002: Disconnetted

Thu Sep 26 18:14:20 2002: Connected to server
Thu Sep 26 18:14:23 2002: Disconnetted

Thu Sep 26 18:14:23 2002: 10 games saved
```

La funzione di log viene attivata col comando

```
log on
```

e disattivata col comando

```
log off
```

Il valore di default è `ON` (è possibile modificare tale valore dal file `pgn-saver/source/h/def.h`).

Il file di log viene salvato nella stessa directory dove si trova l’eseguibile e si chiama `pgns.log`.

L’utente può decidere se sovrascrivere di volta in volta il file di log o se aggiungere l'output alla fine del file.

Se si preferisce la prima opzione si inserisce il comando

```
log overwrite
```

In questa modalità il file di log viene sovrascritto ogni volta che il programma viene eseguito.

Attenzione: esecuzione non significa connessione. Il file di log viene sovrascritto ad ogni esecuzione ma non ad ogni connessione.

Se per esempio il programma viene eseguito dopo aver inserito il valore 20 per il comando `loop` il file di log sarà relativo a tutta l'esecuzione, cioè a tutte e 20 le connessioni. Se invece lanciamo il programma 20 volte con `loop` impostato a 1 il file di log verrà ogni volta sovrascritto e l'ultima versione conterrà i riferimenti solo alla ventesima connessione.

Per non sovrascrivere il file di log e appendere l'output alla fine del file c'è il comando

```
log append
```

che è l'opzione di default.

9.3.9 Help e valori di default

Il comando

```
help
```

può essere impartito all'interprete per avere, per ogni comando, una breve descrizione sul significato e sulla sintassi.

Il comando

```
help command
```

fornisce, per il comando *command*, informazioni più dettagliate sull'uso, la sintassi e sull'eventuale valore di default.

Il comando

```
default
```

visualizza i valori di default di tutte le opzioni.

9.3.10 PGN-Saver e il demone cron

PGN-Saver può da solo fungere da demone, abbiamo visto come fare quando abbiamo descritto i comandi `loop` e `interval`.

È comunque possibile usarlo congiuntamente al demone *cron*, per decidere data e ora di inizio o anche per demandare a tale programma la gestione dei tempi e degli intervalli, invece di usare i comandi `loop` e `interval`, vedremo in seguito come.

Per editare il proprio file `crontab` si usa il comando

```
crontab -e
```

da impartire al prompt della shell del sistema operativo.

Per l'editing di un file `crontab` si rimanda alla voce `crontab` del manuale di Unix, sessione cinque. Per informazioni più generali sui file `crontab` vedere la stessa voce alla sessione uno. Per informazioni sul demone cron digitare `man cron`.

La riga di comando nel file `crontab` deve specificare il percorso e il nome dell'eseguibile. A tal proposito PGN-Saver è stato progettato per accettare il comando `start` dalla linea di comando, in questo modo

```
./pgns start
```

Inserendo questo comando viene saltata l'esecuzione dell'interprete di comandi e quindi viene invocato `start` senza attendere che l'utente inserisca altri comandi al prompt.

Supponendo che l'eseguibile si trovi nella directory `pgn-saver`, contenuta nel primo livello della home directory dell'utente, e supponendo che l'utente voglia far partire PGN-Saver alle ore 12 del primo di ogni mese, la riga di comando da inserire nel file `crontab` sarà

```
00 12 1 * *     $HOME/pgn-saver/./pgns start
```

Come dicevamo prima, la gestione delle connessioni può essere interamente demandata al demone cron impostando il valore 1 per il comando `loop` (cioè viene effettuata una sola connessione) e editando il file `crontab` di conseguenza.

Per esempio per eseguire il programma ogni ora impostare 1 per il comando `loop` e quindi editare il file `crontab` come segue:

```
00 * * * *      $HOME/pgn-saver/./pgns start
```

Naturalmente, dal momento che PGN-Saver accetta soltanto il comando `start` come argomento alla linea di comando, le altre opzioni vanno modificate intervenendo manualmente sui valori di default nel file `pgn-saver/source/h/def.h`.

Per visualizzare il proprio file `crontab` digitare

```
crontab -l
```

al prompt della shell del sistema operativo; per rimuovere il file usare il comando

```
crontab -r
```

Attenzione: può capitare che usando il demone cron sia impostato per default che l'apertura e la creazione di file avvenga in una determinata directory. Per esempio può capitare che l'eseguibile si trovi nella directory `$HOME/pgn-saver` ma utilizzando il demone cron il file `pgns.log` venga creato nella directory `$HOME` e, soprattutto, il file `pgns.pls` venga cercato nella stessa directory `$HOME` e quindi non trovato, generando un errore.

Nell'editare il file `crontab` è possibile modificare il valore della directory `$HOME`, inserendo la riga

```
HOME = value
```

In alternativa è comunque possibile spostare l'eseguibile `pgns` e il file `pgns.pls` a livello della directory in cui vengono aperti e creati i file, nel nostro esempio la directory `$HOME`, in modo da evitare qualsiasi complicazione.

Quando PGN-Saver viene lanciato dal demone cron, la modalità di log viene attivata in ogni caso, indipendentemente dal valore di default. Questo perché utilizzando il demone cron non si ha la disponibilità di un terminale su cui stampare i messaggi, che vengono di conseguenza fatti tutti convogliare sul file di log.

I messaggi che normalmente verrebbero stampati a video vengono inibiti (se così non fosse l'utente si vedrebbe recapitare una mail dal demone cron con una stampa di tutti i messaggi a video prodotti).

Inoltre, quando si usa il software congiuntamente al demone cron, la sola modalità di input utilizzabile è quella da file e la scrittura sul file di log avviene in modalità `APPEND`.

Le opzioni che è previsto possano essere modificate manualmente dall'utente quando si usa PGN-Saver con il demone cron sono

- `type (pgn-saver/source/c/main.c)`,
- `loop (pgn-saver/source/h/def.h)`,
- `interval (pgn-saver/source/h/def.h)`,
- `ntry (pgn-saver/source/h/def.h)`,
- `remove (pgn-saver/source/h/def.h)`.

Per tutte le altre viene effettuato l'overriding con le opzioni imposte per l'utilizzo di cron.

Da notare che PGN-Saver può essere lanciato con il comando `start` come argomento alla linea di comando anche manualmente, senza l'utilizzo del demone cron. Se questo avviene rimangono comunque valide le condizioni imposte viste precedentemente, cioè

- il file di log viene sempre creato,
- il file di log viene scritto in modalità `APPEND`,
- il terminale non viene utilizzato,
- la modalità di input è quella da file.

9.3.11 Come uscire dal programma

Un modo per uscire dal programma è quello di impostare un numero massimo di connessioni (vedi par. 9.3.6).

È possibile anche inviare un interrupt con la combinazione di tasti CTRL+C per interrompere il programma in qualunque punto si trovi. Il programma effettua il catch del segnale, stampando il resoconto delle connessioni (cioè stampando il numero totale di file salvati correttamente) e uscendo in modo non anomalo.

È consigliabile usare questo comando soltanto tra una connessione e l'altra, cioè durante l'intervallo in cui il programma non fa niente. Se l'interrupt viene inviato nell'arco di tempo in cui una connessione è attiva il salvataggio dei file può non avvenire in modo corretto.

Se l'utente desidera uscire dall'interprete dei comandi ancor prima di aver lanciato il comando `start` può farlo inserendo il comando

```
exit
```

Tale comando può essere inserito solo da prompt: non può essere usato una volta che il programma ha iniziato il ciclo di connessioni.

9.4 I nomi dei file

Le partite vengono salvate una per file. Il nome di ogni file è così composto:

user-opponent.month.day.year.h:m.pgn

dove *user* è uno degli utenti specificati da un'handle, *opponent* è l'avversario di *user* per quella partita, *month*, *day* e *year* rappresentano la data e *h:m* l'ora.

Per esempio per una gara giocata tra hazel e SilverStar il 26 settembre 2002 e conclusasi alle ore 17:24 verrà salvato un file con il nome

```
hazel-SilverStar.Sep.26.02.17:24.pgn
```

9.5 Le basi del programma

Il programma si basa su tre componenti fondamentali:

- output del comando *history*,
- output del comando *smoves*,
- parsing dell'output del server (datagram di livello 1).

9.5.1 Il comando *history*

Il comando *history* visualizza i riferimenti alle 20 partite più recenti giocate da un determinato giocatore. La sintassi di tale comando è

```
history user
```

Ecco un esempio dell'output del comando *history* per il giocatore KillerQN:

```
Recent games of KillerQN:

                   Opponent     Type         ECO End Date
12: - 1743 W 1751 SilverStar   [16r  3   3] --- WQ  Sep 12 02 19:23
11: + 1760 B 1716 PepperSpray  [16r  3   3] --- WQ  Sep 12 02 18:57
10: + 1434 B 1401 MatingU      [ sr  5   0] B30 Res Sep 12 02 18:46
 9: + 1738 B 1777 Chisel       [ 4r  4   4] --- Res Sep 12 02 18:08
 8: + 1718 W 1676 swingmaster  [16r  3   3] --- Res Sep 12 02 13:12
 7: - 1702 B 1692 swingmaster  [16r  3   3] --- Mat Sep 12 02 13:06
 6: = 1720 W 1708 SirEbbal     [16r  3   3] --- NM  Sep 12 02 12:42
 5: + 1919 W 1725 mate         [ sr  9   9] C40 Res Sep 12 02 11:31
 4: = 1721 B 1738 diswin       [16r  3   3] --- NM  Sep 12 02 11:17
 3: = 1720 W 1739 diswin       [16r  3   3] --- Sta Sep 12 02 11:09
 2: - 1719 B 1887 MeisterZinge [16r  3   3] --- Res Sep 12 02 10:20
 1: - 1728 B 1891 Onulet       [16r  3   3] --- Fla Sep 12 02 09:46
 0: - 1738 B 1732 pklm         [ 4r  4   4] --- Mat Sep 12 02 08:43
99: + 1756 W 1714 pklm         [ 4r  4   4] --- Res Sep 12 02 08:32
98: + 1740 W 1563 TopalovFan   [16r  3   3] --- Res Sep 12 02 08:10
97: + 1736 B 1591 TopalovFan   [16r  3   3] --- Res Sep 12 02 08:05
96: + 1732 W 1628 TopalovFan   [16r  3   3] --- Mat Sep 12 02 07:54
95: + 1728 B 1678 TopalovFan   [16r  3   3] --- Mat Sep 12 02 07:51
94: - 1528 W 1400 odysseus     [ fr  5   0] A00 Fla Sep 12 02 07:40
93: = 1551 B 1512 ZenChess     [ fr  5   0] B41 NM  Sep 12 02 06:23
```

Le partite vengono numerate ciclicamente da 0 a 99. I campi *Opponent* e *Date* vengono usati per formare i nomi dei file salvati. Il campo *Type* è confrontato col tipo

specificato dall'utente per salvare o meno la partita. La seconda colonna (`'+'` sta per vittoria, `'-'` per sconfitta e `'='` per patta) viene letta per escludere le partite aborted (`'a'`).

Il programma copia la tabella in un buffer temporaneo e la usa per stabilire quali partite vanno salvate.

9.5.2 Il comando *smoves*

Il comando *smoves* visualizza le mosse di una partita. La sintassi è la seguente

```
smoves user number
```

dove *number* è il numero della partita che si vuole visualizzare, preso tra uno dei 20 numeri a disposizione nella tabella di history del giocatore *user*.

Ecco un esempio dell'output del comando

```
smoves killerqn 10
```

```
MatingU (1401) vs. KillerQN (1434) --- 2002.09.02 18:46:29
Rated 5-minute match, initial time: 5 minutes, increment: 0 seconds

Move  MatingU            KillerQN
----  ----------------   ----------------
  1.  e4       (0:01)    c5       (0:02)
  2.  Nf3      (0:01)    Nc6      (0:02)
  3.  Bb5      (0:01)    Nd4      (0:04)
  4.  Nxd4     (0:01)    cxd4     (0:01)
  5.  O-O      (0:01)    a6       (0:03)
  6.  Be2      (0:02)    d5       (0:04)
  7.  e5       (0:03)    f6       (0:04)
  8.  Bh5+     (0:17)    g6       (0:01)
  9.  exf6     (0:04)    Nxf6     (0:05)
 10.  Bf3      (0:04)    e5       (0:02)
 11.  d3       (0:01)    e4       (0:04)
 12.  dxe4     (0:01)    dxe4     (0:01)
 13.  Be2      (0:01)    Bf5      (0:03)
 14.  Bg5      (0:01)    Bc5      (0:05)
 15.  g4       (0:04)    Be6      (0:07)
 16.  Re1      (0:09)    O-O      (0:38)
 17.  Nd2      (0:08)    Bd5      (0:06)
 18.  b3       (0:15)    e3       (0:14)
 19.  fxe3     (0:11)    dxe3     (0:03)
 20.  Nc4      (0:05)    Bxc4     (0:25)
 21.  Bxc4+    (0:04)    Kg7      (0:01)
```

```
22.  Qf3      (0:17)    Qd7      (0:11)
23.  h3       (0:19)    Rae8     (0:03)
24.  Qf4      (0:05)    Ng8      (0:08)
25.  Qg3      (0:23)    Qd2      (0:07)
26.  Rad1     (0:06)    Qxd1     (0:14)
27.  Rxd1     (0:08)    e2+      (0:01)
28.  Kg2      (0:10)    Rf2+     (0:11)
      {White resigns} 0-1
```

Il programma, quindi, per ogni giocatore specificato dall'utente invia il comando *history*, e per le sole partite nuove (e non aborted) che fanno match con il tipo specificato dall'utente invia il comando *smoves* e le salva su file.

9.5.3 Datagram di livello 1

Come abbiamo visto nella sezione 8.10 l'output di ICC è di due livelli: il livello 1 e il livello 2. Per gli scopi di PGN-Saver il livello 1 è sufficiente: si tratta infatti di leggere l'output dei comandi *history* e *smoves*, che sono contemplati da quel tipo di datagram.

Al momento della connessione, quindi, l'agente invia al server la stringa

```
set level1 1
```

e il server risponde con il messaggio

```
level1 set to 1
```

Un datagram di livello 1, come già spiegato nel capitolo precedente, ha la forma

^Y[*command_number player* "command_output" ^Y]

Quando il programma, leggendo dall'output del server, incontra il carattere `^Y`, guarda di che tipo è il carattere successivo: se si tratta di un carattere `OPEN1` (`[`) sa che sta per iniziare un nuovo blocco, se invece incontra `CLOSE1` (`]`) capisce che il blocco è terminato.

Naturalmente il programma è in grado di effettuare il parsing corretto anche dei blocchi annidati. Per esempio se viene ricevuto il carattere `^Y`, seguito da `OPEN1`, non è detto che dopo il contenuto del datagram si incontrerà la sequenza `^Y]`, potrebbe

infatti capitare di trovarsi di fronte a una nuova sequenza `^Y[`, e quindi ad un nuovo datagram, annidato all'interno del primo. Il programma effettua il parsing solo a livello zero, ovvero quando i datagram annidati sono stati tutti chiusi.

Come già spiegato, *`player`* contiene il riferimento al giocatore che ha inviato il comando al quale l'output si riferisce. Gli output trattati dal programma per il salvataggio di partite saranno conseguenza di comandi inviati dal programma stesso e quindi il valore di *`player`* sarà in questi casi `*` (significa appunto che l'utente da cui è giunto il comando e a cui viene inviato il datagram coincidono).

Infine, il valore *`command_number`* è essenziale per stabilire per ogni datagram a quale comando si riferisce. Nel nostro caso i codici che ci interessano sono

```
#define CN_HISTORY                    129
#define CN_SMOVES                     171
```

I datagram con `command_number = 129` si riferiscono al comando *history*: il loro output va salvato in un buffer temporaneo usato per stabilire quali partite andranno salvate. Ecco come viene ricevuta in realtà la tabella del paragrafo 9.5.1:

```
^Y[ 129 * \nRecent games of KillerQN:\n\n                    Opponent
Type          ECO End Date\n12: - 1743 W 1751 SilverStar    [16r  3
3] --- WQ  Sep 12 02 19:23\n11: + 1760 B 1716 PepperSpray   [16r  3
3] --- WQ  Sep 12 02 18:57\n10: + 1434 B 1401 MatingU       [ sr  5
0] B30 Res Sep 12 02 18:46\n 9: + 1738 B 1777 Chisel        [ 4r  4
4] --- Res Sep 12 02 18:08\n 8: + 1718 W 1676 swingmaster   [16r  3
3] --- Res Sep 12 02 13:12\n 7: - 1702 B 1692 swingmaster   [16r  3
3] --- Mat Sep 12 02 13:06\n 6: = 1720 W 1708 SirEbbal      [16r  3
3] --- NM  Sep 12 02 12:42\n 5: + 1919 W 1725 mate          [ sr  9
9] C40 Res Sep 12 02 11:31\n 4: = 1721 B 1738 diswin        [16r  3
3] --- NM  Sep 12 02 11:17\n 3: = 1720 W 1739 diswin        [16r  3
3] --- Sta Sep 12 02 11:09\n 2: - 1719 B 1887 MeisterZinge [16r  3
3] --- Res Sep 12 02 10:20\n 1: - 1728 B 1891 Onulet        [16r  3
3] --- Fla Sep 12 02 09:46\n 0: - 1738 B 1732 pklm          [ 4r  4
4] --- Mat Sep 12 02 08:43\n99: + 1756 W 1714 pklm          [ 4r  4
4] --- Res Sep 12 02 08:32\n98: + 1740 W 1563 TopalovFan    [16r  3
3] --- Res Sep 12 02 08:10\n97: + 1736 B 1591 TopalovFan    [16r  3
3] --- Res Sep 12 02 08:05\n96: + 1732 W 1628 TopalovFan    [16r  3
3] --- Mat Sep 12 02 07:54\n95: + 1728 B 1678 TopalovFan    [16r  3
3] --- Mat Sep 12 02 07:51\n94: - 1528 W 1400 odysseus      [ fr  5
0] A00 Fla Sep 12 02 07:40\n93: = 1551 B 1512 ZenChess      [ fr  5
0] B41 NM  Sep 12 02 06:23\n ^Y]
```

I datagram con `command_number = 171` si riferiscono al comando *smoves*: l'output relativo va salvato in un file (in formato PGN). I datagram con codice diverso

dai due precedenti, che possono comunque arrivare, (per esempio messaggi di benvenuto, risultato del settaggio del livello 1, messaggi in fase di logout, o anche messaggi personali inviati da altri utenti o messaggi diretti a tutti gli utenti che si trovano in un determinato canale) vengono letti, ma non viene intrapresa nessuna particolare azione dopo la loro ricezione.

Il programma effettua correttamente anche il parsing dei datagram di livello 2, nel caso dovesse giungerne qualcuno. I datagram di livello 2 vengono identificati dalle sequenze `^Y(` e `^Y)` (i caratteri speciali "(" e ")" vengono associati dal programma alle costanti `OPEN2` e `CLOSE2`).

9.6 Il dialogo col server

Il dialogo col server avviene aprendo un socket e connettendosi all'host `204.178.125.65`, usando la porta numero 5000.

I comandi dal client al server vengono inviati attraverso la funzione `sendToICC`, che scrive una stringa sul descrittore usato per la comunicazione. I dati dal server al client vengono letti attraverso la funzione `read`, che riempie ad ogni lettura un buffer della dimensione massima di 1000 caratteri. Ogni volta che viene eseguita una lettura inizia il parsing dei dati appena letti. Naturalmente non è assolutamente detto che i dati letti corrispondano esattamente a un datagram o ad un multiplo di datagram. Infatti, essendo il buffer di 1000 caratteri, risulterà quasi sempre che vengano lette anche porzioni di datagram.

Vediamo, quindi, a partire dal termine di una lettura dal descrittore del server, tutte le azioni che il programma intraprende per effettuare il parsing dell'output.

- Funzione `parse`: la funzione `parse` viene invocata appena l'operazione di lettura è terminata[2], ed ha come parametro il buffer riempito con i dati letti. Questa funzione non fa nient'altro che scorrere il buffer carattere per carattere, chiamando, per ognuno di essi, la funzione `parseChar`.

- Funzione `parseChar`: quest'altra funzione esamina ogni singolo carattere. Innanzitutto pone il carattere in un altro buffer (il buffer iniziale dev'essere

[2] Ciò non significa necessariamente che il server ha esaurito l'output, ma soltanto che il buffer da noi destinato per la lettura è stato riempito.

immediatamente disponibile per un'ulteriore lettura). Se tale buffer è pieno aggiunge le stringhe lette ad una lista, di dimensione illimitata, dove vengono parcheggiate tutte le stringhe lette ma non ancora trattate.

A questo punto viene controllato se il carattere precedente era un `^Y`; se così è stato si controlla qualora il carattere corrente sia un `OPEN1` (`[`) o un `CLOSE1` (`]`): nel primo caso si aumenta il livello di nesting dei datagram, nel secondo caso lo si diminuisce. Nel caso in cui abbiamo ricevuto un `CLOSE1` e il livello di nesting scende a zero viene finalmente eseguito il parsing vero e proprio invocando la routine `parseBuffer` con parametro la lista illimitata più il buffer corrente: viene cioè eseguito il parsing di tutto l'output che si ha in memoria e che non è stato ancora trattato.

- Funzione `parseBuffer`: è il primo passo verso il parsing dell'output memorizzato nella lista. Essa ignora il primo carattere (un carattere pronto per il parsing, preso dalla lista, è denominato *token*), che deve essere il `^Y` e controlla che da quel punto in poi ci sia un `OPEN1` (deve esserci, dal momento che il parsing viene effettuato quando si raggiunge il livello zero di annidamento; ciò significa che la lista viene svuotata solo una volta che i datagram sono completi e quindi sarà riempita solo di nuovi datagram, che quindi iniziano con `OPEN1`. In sostanza, la ricerca dell'`OPEN1` effettuata a questo punto, serve, oltre che per assicurarsi che non ci siano errori, per saltare eventuali spazi inseriti tra `^Y` e `[`). Una volta terminato con successo tale controllo viene chiamata la funzione `level1Block`.

- Funzione `level1Block`: come prima cosa invoca `getWord`, che legge dal buffer tutto quello che va dall'`OPEN1` appena trovato al prossimo carattere speciale (delimitatore di datagram). Ritorna in questo modo tutto quello che è contenuto in un datagram. Il controllo ritorna a `level1Block`, che da tale insieme di caratteri legge i valori di `command_number` e `player`.

A questo punto se il token corrente è un `CLOSE1` vengono settate alcune variabili di controllo che informano che il datagram è terminato; se si tratta di un nuovo `OPEN1` la funzione chiama se stessa ricorsivamente con parametro il livello di annidamento aumentato di uno.

E veniamo al punto che più ci interessa ai fini degli scopi del programma. Se infatti il token non è né un `OPEN1` né un `CLOSE1`, ma si tratta di un carattere normale, si controlla il codice del datagram per sapere a quale comando risponde. Se il

comando inviato era *history* significa che il carattere fa parte della tabella history (vedi paragrafo precedente) e quindi viene aggiunto al buffer temporaneo che sarà poi usato per il salvataggio delle partite. Se il codice corrisponde al comando *smoves* il carattere fa parte dell'output che va salvato in un file PGN e viene quindi invocata una system call `write` che scrive il carattere sul descrittore di file attualmente in uso. Qualsiasi altro carattere viene letto, tolto dal buffer, ma viene poi eliminato.

- Funzione `getToken`: prende di volta in volta un carattere dal buffer in attesa di parsing e controlla che non sia un `EOF` (non dovrebbe mai accadere a meno di errori).

Menzioniamo infine, per la comunicazione da client a server, la funzione `login`, invocata subito dopo la connessione. La stringa inviata come login è `guest`, cioè si sfrutta la possibilità di poter entrare come ospiti senza necessità di possedere un account e quindi di dover inserire una password.

9.7 La tabella history

La tabella che viene inviata dal server come output del comando *history* (per un esempio vedi paragrafo 9.5.1) viene letta e memorizzata in un buffer temporaneo. Questo buffer verrà in seguito letto ogni volta che serviranno certe informazioni. Queste sono le funzioni preposte a tale compito:

- `getFirstGame` e `getNextGame` ritornano una stringa contenente rispettivamente il primo numero di partita disponibile (dove per primo si intende il più alto partendo dall'inizio del file) e il prossimo numero di partita a partire da quello corrente. Tali funzioni vengono invocate per sapere quale partita si deve salvare e poter lanciare così il comando `smoves` *`user number`*.

- `getOpponent` e `getDataAndTime` estraggono il nome dell'avversario dal campo *Opponent* e la data e l'ora dal campo *Date* per comporre il nome del file da salvare.

- `wantedType` verifica che il campo *Type* della tabella corrisponda al tipo di partita specificato dall'utente. Qualora i due tipi differiscano la partita viene ignorata e si passa alla successiva.

• `abortedGame` esclude dal salvataggio le partite aborted (cioè che contengono il carattere `'a'` sulla seconda colonna della tabella).

9.8 Nomi di file: concatenazione

La funzione `concatenateToFilename` concatena la stringa `filename` alla stringa passata come parametro e memorizza il risultato nuovamente in `filename`. I nomi dei file vengono quindi formati per concatenazioni successive:

- parte dalla stringa contenente la handle del giocatore,
- concatena il carattere `'-'`,
- concatena il risultato della funzione `getOpponent`,
- concatena il carattere `'.'`,
- concatena il risultato della funzione `getDataAndTime`,
- concatena la stringa `'.pgn'`.

9.9 File di log: creazione e scrittura

Come già detto il file di log viene creato solo se il log mode è attivo e l'output viene sovrascritto se l'opzione `overwrite` è settata oppure aggiunto alla fine del file se viene specificata l'opzione `append`. Tale meccanismo è stato semplicemente realizzato attraverso l'istruzione

```
open("pgns.log", O_RDWR | O_CREAT | log_output_flag, S_IRUSR |
      S_IWUSR | S_IXUSR)
```

Dove `log_output_flag` assume il valore `O_TRUNC` in presenza dell'opzione `overwrite` e `O_APPEND` in modalità `append`.

Il file viene creato se non esiste (flag `O_CREAT`).

Il file di log viene aggiornato di volta in volta invocando la funzione `writeLog` (o anche la funzione `writeLogAndTerm` che, oltre ad aggiornare il file di log stampa

il messaggio a video). La routine provvede a stampare ora e data dell'aggiornamento seguite dalla stringa passata come parametro.

9.10 Metodo per il salvataggio delle partite

Abbiamo visto che a livello più basso il salvataggio delle partite avviene all'interno della funzione `level1Block`, in cui, una volta esaminato un singolo carattere, se questo fa parte dell'output del comando *smoves* (cioè il codice del comando del datagram corrente è 171) viene stampato nel file a cui l'attuale file descriptor in uso si riferisce.

Vediamo in questo paragrafo come avviene la creazione di tale file e in particolare come è stato realizzato il metodo per far sì che l'agente salvi esclusivamente le partite nuove tra una connessione e l'altra.

Una volta calcolato il nome del file attraverso il meccanismo della concatenazione (vedi par. 9.8), il file viene aperto con l'istruzione

```
open(filename, O_WRONLY | O_CREAT | O_EXCL, S_IRUSR | S_IWUSR |
     S_IXUSR)
```

Il flag `O_CREAT` fa sì che il file venga creato se al momento dell'apertura non esiste e questo è ciò che dovrebbe accadere se la partita che stiamo per salvare non è stata già salvata in precedenza. Il flag `O_EXCL` genera un errore qualora il file esista già e il flag `O_CREAT` sia stato settato.

Accade quindi che per ogni partita viene calcolato il nome del file attraverso concatenazione. Fatto questo si tenta di aprirlo con l'istruzione precedente: se la `open` non genera un errore significa che il file non esiste e quindi la partita va salvata; se invece ritorna `-1` vuol dire che la partita è già stata salvata in una precedente connessione e il programma passa direttamente alla partita successiva.

Questo metodo è stato scelto perché molto semplice nonché veloce. Nonostante il calcolo del nome del file venga eseguito anche per le partite che poi non verranno salvate è comunque più rapido di qualsiasi altro controllo volto a stabilire se una partita è già stata salvata: controlli sul numero delle partite o sulla data che andrebbero effettuati sul buffer dove è memorizzato l'output di *history*.

10 Tool

10.1 Viewer e editor

I programmi di visualizzazione e editing di file EPD e PGN permettono di:

- visualizzare partite in formato EPD e PGN,
- editare posizioni,
- salvare in formato EPD o PGN,
- stampare,
- creare diagrammi in formato GIF, JPG, BMP e anche HTML, che possono essere esportati per altri usi e applicazioni.

Alcuni editor permettono inoltre di caricare posizioni, scegliere un motore scacchistico che supporta la funzionalità di *analyze* (specificando il percorso dell'eseguibile) ed effettuare l'analisi con la finestra dell'editor.

Ecco qualche indirizzo dove si possono trovare editor e visualizzatori gratuiti o in versione demo:

Π Chess Captor – *http://www.migmag.pair.com/captor/*

Π Chess Diagrams – *http://www.alphaprime.com/chessmaker/showcase*

Π Ecw Chess – *http://ecwchess.narod.ru/Index.html*

Π EPD2diag – *http://www.rebel.nl/epd2diag.htm*[1]

Π ChessBrain Annotator – *http://www.chessbrain.de*

Π Chess Geek – *http://hometown.aol.com/ChessMonkeys/Software.htm*

Π Mini-PGN – *ftp://sunsite.srce.hr/pub/misc/chess/minipgn2.zip*

Π PGNread – *http://scacchi.qnet.it/software/download/pgnread.zip*

Π PGNEdit – *http://www.sihope.com/~ponstad/utilities/PGNEdit.zip*

Ecco due tool specifici per la visualizzazione PGN su pagine web:

Π J!ChessViewer – *http://www.geocities.com/yccheok/jchess/index.html*

Π Montreaux – *http://www.jphendriks.myweb.nl/montreux/index.html*

Con quest'altro è possibile creare diagrammi animati per pagine web :

Π PGNtoJS – *http://www.mailchess.de/engl/indexe.html*

Esistono anche pagine contenenti Applet Java in grado di generare semplici diagrammi che possono essere esportati verso altre applicazioni.

Π Jdiag – *http://www.trojanco.demon.co.uk/jdiag/JDiag.html*

Si possono inoltre scaricare piccole macro per Microsoft Word; sono molto interessanti perché occupano pochissimo spazio e consentono di produrre diagrammi molto simili a quelli dei più comuni editor senza dover prima generare immagini da esportare, ma semplicemente editando direttamente dal documento Word.

Π FEN2diag – *http://www.chessvariants.com/d.font/fonts.html*

Π PGN2fig – *http://www.chessvariants.com/d.font/fonts.html*

[1] È il tool con il quale sono stati realizzati i diagrammi del capitolo 2.

Con questo tool è possibile generare documenti Latex, PostScript e HTML contenenti diagrammi:

Π FEN2Latex – *http://www-user.rhrk.uni-kl.de/~wehner/fen2latex.html*

10.2 Database management

Un tipo di software che negli ultimi anni ha avuto sempre più successo tra gli appassionati di scacchi sono i database. Le funzioni più comuni di questi tool sono:

- archiviazione di partite,
- gestione di database di giocatori,
- caricamento, editing e salvataggio di partite,
- stampa,
- possibilità di rivisitare le partite, navigando attraverso le mosse (animazioni sulla scacchiera, liste di mosse, alberi),
- analizzare una partita con un motore scacchistico che supporta tale funzione,
- possibilità di effettuare ricerche di partite specificando i parametri voluti (data, giocatore, risultati finali; è inoltre possibile combinare i vari parametri secondo le operazioni classiche che agiscono sui database: unione, intersezione, differenza, …).

Questi sono alcuni database management freeware o shareware che si possono trovare in rete:

Π CDB – *http://scacchi.qnet.it/software/download/cdb.zip*

Π Chess Assistant – *http://store.convekta.com/shop_model.asp*

Π Chess Pad – *http://www1.tip.nl/~t799997/chesspad.htm*

Π Scid – *http://scid.sourceforge.net/*

Π YACD – *http://starship.python.net/crew/adjih/data/yacd/yacd/Doc/yacd.html*

Π Slinger – *http://www.sihope.com/~ponstad/utilities/Slinger.zip*

Π ExaChess – *http://www.exachess.com/*

10.3 Scacchi via e-mail

Abbiamo già visto nel paragrafo 6.6 le caratteristiche di un semplicissimo programma che gestisce partite di scacchi via e-mail.

Di seguito sono indicati alcuni indirizzi dove si possono trovare programmi per scacchi via e-mail, gratuiti o in versione demo, che utilizzano gradevoli interfacce grafiche e hanno qualche funzione in più rispetto a CMail.

Π Chess Recorder – *http://www.halcyon.com/ericch/chess.htm*

Π Chess Tool – *http://www.pasanet.es/usuarios/jgajate/*

Π Chess Edt – *ftp://ftp.pitt.edu/group/student-activities/chess/UTIL/cedt70.zip*

Π DBS Chess – *http://www.dbsugden.clara.net/where.htm*

Π ECTool – *http://www.ectool.nu/*

Π MailChess – *http://www.mailchess.de/engl/indexe.html*

10.4 Altri tool

Vediamo in quest'ultimo paragrafo altri tool di vario tipo.

Con questi piccoli programmi è possibile calcolare il punteggio ELO di un giocatore:

Π Chess Calculator – *http://www.echessbook.com/products/prod0021.htm*

Π ELO Calculator – *http://www.geocities.com/elocalculator/*

Questi altri sono gestori di tornei tra più giocatori:

- Diena – *http://www.infcom.it/fsi/diena.html*
- Swiss Perfect – *http://www.swissperfect.com/*

Il seguente è un risolutore di problemi scacchistici:

- MateMaster – *http://www.matemaster.de/*

Questo invece è un database di problemi scacchistici:

- Tactical Database – *http://www.mailchess.de/engl/indexe.html*

Un sito con un'ampia gamma di link a tool scacchistici è

- *http://www.szachy.pl/programy/*

Infine è possibile scaricare fonts per interfacce grafiche all'indirizzo

- *http://www.chessvariants.com/d.font/fonts.html*

11 Conclusioni

11.1 Scacchi e computer: conclusioni

Abbiamo tentato di spiegare come mai gli scacchi sono un gioco molto studiato per quanto riguarda l'applicazione al calcolatore. Siamo poi entrati dentro un motore scacchistico, analizzando tutti quei metodi, tecniche e strategie, consolidati nel corso degli anni, che costituiscono un buon programma per scacchi.

Abbiamo visto quali sono le caratteristiche di una interfaccia grafica, un tipo di software usato da tutti i giocatori di scacchi al computer, e descritto gli Internet Chess Server, dai servizi offerti fino ai dettagli del protocollo di comunicazione.

Il nostro intento è stato quello di mostrare sia i miglioramenti a cui sono giunti i motori scacchistici, sia come il panorama del software scacchistico non si limiti ai giocatori artificiali, ma comprenda una vastissima serie di strumenti e servizi, tra cui gli ormai diffusissimi Internet Chess Server.

11.2 PGN-Saver: commenti sul lavoro svolto

Una delle parti più interessanti e stimolanti è stata proprio l'analisi degli Internet Chess Server: in particolare il modo in cui dialogano con il client, il formato dei dati in output, le particolarità del protocollo di comunicazione. Il software PGN-Saver ne è un piccolo esempio.

Oltre a questo, con la realizzazione del programma, abbiamo dimostrato che gli Internet Chess Server non sono solo luoghi virtuali nei quali è possibile giocare a scacchi, ma mettono a disposizione numerosi servizi. Abbiamo inoltre dato un esempio di come gli Internet Chess Server siano anche fonti di informazioni.

Il programma, che conta 2575 righe di codice (commenti compresi) e 13 file, è stato realizzato in linguaggio C, su sistema operativo Linux Red Hat 7.0. La macchina utilizzata monta un processore AMD-K7 a 500 MHz e dispone di 128 MB di RAM.

11.3 Prospettive future

Abbiamo visto come, soprattutto sulla rete, il gioco degli scacchi abbia trovato uno spazio davvero rilevante. Sono tantissimi i siti che permettono di scaricare software di tutti i tipi, ne abbiamo avuto un assaggio nel capitolo dieci. Non esiste un servizio legato al gioco degli scacchi che non sia stato tradotto in un programma.

Dal punto di vista delle applicazioni, soprattutto via rete, il futuro appare quindi tutto da scoprire. Non sono molti anni, infatti, che abbiamo assistito alla diffusione di interfacce grafiche e tool vari.

Per quanto riguarda i giocatori artificiali, invece, il discorso cambia. Se un tempo l'informatico appassionato di scacchi si concentrava prevalentemente sui dettagli dei motori scacchistici, oggigiorno sono sempre più numerosi i progettisti di interfacce grafiche, gli sviluppatori di tool sempre più complessi e completi, i fondatori di Internet Chess Server.

D'altra parte è poco probabile che le tecniche e gli algoritmi che governano i programmi per scacchi vengano migliorati sensibilmente; lo dimostra il fatto che i progettisti di calcolatori come Deep Blue basano principalmente le loro ricerche sull'hardware. Sembra, in sostanza, che a livello algoritmico si sia già fatto tutto o quasi e quindi le speranze di avere programmi sempre più forti ricadono sull'hardware.

Questa mancanza di prospettive di sviluppo nel campo degli algoritmi classici sta portando molti programmatori ed esperti di Intelligenza Artificiale ad orientarsi verso differenti tipi di strategie: algoritmi genetici o su basi inferenziali, per esempio. Molti si stanno impegnando anche nello studio di giochi a informazione incompleta, cioè

non risolvibili meccanicamente data la mancanza di alcune informazioni sullo stato del gioco.

Abbiamo visto nel capitolo introduttivo che scacchi e computer vanno d'accordo proprio per il fatto che gli scacchi sono sì un gioco ad informazione completa, ma troppo complesso per essere risolto algoritmicamente; è questo a suscitare interesse nella comunità informatica. Nessuno ci impedisce di pensare che un giorno i calcolatori avranno una tale potenza di calcolo da poter creare un computer in grado di giocare la partita perfetta, ossia risolvere il gioco degli scacchi in un tempo finito nonché ragionevolmente breve. Sembra impossibile, e probabilmente lo è, se solo pensiamo che attualmente anche il computer più potente per giocare una tale partita impiegherebbe molti miliardi di anni. D'altronde, fino a cinquant'anni fa, nessuno avrebbe ritenuto possibile produrre computer in grado di arrivare a una velocità dell'ordine dei miliardi di cicli al secondo e con memoria centrale dell'ordine dei centinaia di milioni di byte.

Forse un giorno tutta la teoria dei giochi applicata agli scacchi diventerà inutile; non varrà più la pena di cercare l'algoritmo ottimale per una determinata funzione o la struttura dati accessibile più rapidamente e tutti gli studi saranno resi vani da una potenza di calcolo da sbalordire il più ottimista degli informatici.

Fino a quel giorno si continua a giocare.

APPENDICE
Codice di PGN-Saver

```
#Makefile



#************************#
#                        #
# CONSTANTS DECLARATIONS #
#                        #
#************************#

OBJ = iccfunct.o stringfunct.o cmd.o history_parser.o connection.o main.o

FLAGS = -Wall -c              # compilation flags
CC = gcc                      # compiler c
LD = gcc                      # linker



#************************#
#                        #
# LINKING                #
#                        #
#************************#

pgns: $(OBJ)
	$(LD) -o pgns $(OBJ)



#************************#
#                        #
# FILES COMPILATIONS     #
#                        #
#************************#

main.o: source/c/main.c source/c/iccfunct.c source/h/iccfunct.h source/c/cmd.c
source/h/cmd.h source/c/history_parser.c source/h/history_parser.h source/c/connection.c
source/h/connection.h
	$(CC) $(FLAGS) -o main.o source/c/main.c

iccfunct.o: source/c/iccfunct.c source/h/iccfunct.h source/c/stringfunct.c
source/h/stringfunct.h source/h/def.h
	$(CC) $(FLAGS) -o iccfunct.o source/c/iccfunct.c

stringfunct.o: source/c/stringfunct.c source/h/stringfunct.h
	$(CC) $(FLAGS) -o stringfunct.o source/c/stringfunct.c

cmd.o: source/c/cmd.c source/h/cmd.h source/c/stringfunct.c source/h/stringfunct.h
source/h/def.h
	$(CC) $(FLAGS) -o cmd.o source/c/cmd.c

history_parser.o: source/c/history_parser.c source/h/history_parser.h
	$(CC) $(FLAGS) -o history_parser.o source/c/history_parser.c

connection.o: source/c/connection.c source/h/connection.h source/c/iccfunct.c
source/h/iccfunct.h source/c/cmd.c source/h/cmd.h source/c/history_parser.c
source/h/history_parser.h
	$(CC) $(FLAGS) -o connection.o source/c/connection.c
```

```
/* stringfunct.h - Functions dealing with strings */



#ifndef STRIGNFUNCT_H
#define STRINGFUNCT_H


#include "def.h"

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <ctype.h>


void safeStrcpy(char *u, char *v, int usize);

string_list_t *putStringInList(string_list_t *sl, char *s);

string_list_t *reverseAppendSl(string_list_t *a, string_list_t *b);

string_list_t *reverseSl(string_list_t *sl);

void freeStrings(string_list_t *bp);

void setUpGetOneChar(string_list_t *buf_list);

int getOneChar(void);


#endif
```

```
/* stringfunct.c - Functions dealing with strings */



#include "../h/stringfunct.h"



/*
 ------------------------------------------------------------
|                                                            |
| Copies as much of v into u as it can assuming u is of size |
| usize. Guaranteed to terminate u with a '\0'.              |
|                                                            |
 ------------------------------------------------------------
*/

void safeStrcpy(char *u, char *v, int usize)
{
  strncpy(u, v, usize - 1);
  u[usize - 1] = '\0';
}



/*
 ------------------------------------------------------------
|                                                            |
| Appends the given string at the beginning of the given string |
| list, it returns a ptr to it                               |
|                                                            |
 ------------------------------------------------------------
*/

string_list_t *putStringInList(string_list_t *sl, char *s)
{
  string_list_t *c;

  c = (string_list_t *) malloc(sizeof(string_list_t));
  c->string = (char *) malloc(strlen(s) + 1);
  strcpy(c->string, s);
  c->next = sl;
  return c;
}



/*
 ------------------------------------------------------------
|                                                            |
| Append list b to the reverse of a. Return a pointer to the |
| result.                                                    |
|                                                            |
 ------------------------------------------------------------
*/

string_list_t *reverseAppendSl(string_list_t *a, string_list_t *b)
{
  string_list_t *slx;
  for (; a != NULL; a = slx)
    {
      slx = a->next;
      a->next = b;
      b = a;
    }
```

```
  return b;
}



/*
 ----------------------------------------------------------------
|                                                                |
| Reverse the given list and return a pointer to it.             |
|                                                                |
 ----------------------------------------------------------------
*/

string_list_t *reverseSl(string_list_t *sl)
{
  return reverseAppendSl(sl, NULL);
}



/*
 ----------------------------------------------------------------
|                                                                |
| Free the strings in the list.                                  |
|                                                                |
 ----------------------------------------------------------------
*/

void freeStrings(string_list_t *bp)
{
  string_list_t *xbp;

  for (; bp != NULL; bp = xbp)
    {
      xbp = bp->next;
      free ((char *) bp->string);
      free ((char *) bp);
    }
}



/*
 ----------------------------------------------------------------
|                                                                |
| Initialize getOneChar routine.                                 |
|                                                                |
 ----------------------------------------------------------------
*/

static string_list_t *x_buf_list;
static int x_nbuf;

void setUpGetOneChar(string_list_t *buf_list)
{
  x_buf_list = buf_list;
  x_nbuf = 0;
}
```

```
/*
 ----------------------------------------------------------------
|                                                                |
| Read one a character from x_buf_list.                          |
|                                                                |
 ----------------------------------------------------------------
*/

int getOneChar(void)
{
  int c;

  if (x_buf_list == NULL)
    return EOF;

  c = x_buf_list->string[x_nbuf];
  x_nbuf++;

  /* if the string is completely read go to the next one */
  if (x_buf_list->string[x_nbuf] == '\0')
    {
      x_buf_list = x_buf_list->next;
      x_nbuf = 0;
    }

  return c;
}
```

```
/* cmd.h - Functions dealing with user commands */



#ifndef CMD_H
#define CMD_H


#include "stringfunct.h"
#include "def.h"

#include <time.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>


char start_args[START_ARGS_SIZE];       /* contains arguments of 'start' command */
char handles[START_ARGS_SIZE];          /* contains arguments of 'start' command: it is
                                           used when a wrong handle has to be removed
                                           from file 'pgns.pls' */
int number_start_args;                  /* number of arguments of 'start' command */
char user_string[USER_STRING_SIZE];     /* contains the name of the current user */
int start_input;                        /* from where input arguments are read */
int log;                                /* log mode control */
int log_output;                         /* log_output: overwrite or append */
int interval;                           /* interval time control */
int loop;                               /* loop connections control */
int ntry;                               /* connection tries */
int inserted_start_cmd;                 /* used to break the loop in main */
int fd_log;                             /* file descriptor of log file */
int nrem;                               /* number of handles removed from 'pgns.pls' */
int remotion;                           /* wrong handles remotion control */
int term;                               /* ON if user interacts with terminal, OFF when
                                           something like cron daemon is used */


extern char type[TYPE_SIZE];


void executeCmd(char *str);

void getNextUser(int users);

void displayPrompt(void);

void writeLog(char *str, char *file);

void writeTerm(char *str);

void writeLogAndTerm(char *str);

void removeHandle(void);


#endif
```

```
/* cmd.c - Functions dealing with user commands */



#include "../h/cmd.h"



/*
 ------------------------------------------------------------
|                                                            |
| Called when 'start' command is given. It performs these    |
| tasks:                                                     |
|                                                            |
|  - checks which input mode is on,                          |
|  - if input mode = FILE                                    |
|     * no arguments required,                               |
|     * open and read file 'pgns.pls',                       |
|  - if input mode = KEYBOARD                                |
|     * read from input buffer,                              |
|     * arguments must be given,                             |
|  - copies the arguments in start_args array,               |
|  - counts the number of arguments,                         |
|                                                            |
 ------------------------------------------------------------
*/

void startCmd(char *str)
{
  char *ptr1, *ptr2;
  int arg;                                      /* arguments counting */
  int i, j;
  int players_fd;                               /* fd for 'pgns.pls' */
  char read_buffer[BUFFER_SIZE];                /* for reading 'pgns.pls' */

  arg = 0;

  if (start_input == PLAYERS_FILE)              /* input from file */
    {
      if (strncmp(str, "start ", 6) == 0)
       if ((str[6] != '\n') && (str[6] != '\0'))
         {
           writeTerm("Invalid command: probably input mode set to be from file");
           return;
         }

       if ((players_fd = (open("pgns.pls", O_RDWR, S_IRUSR | S_IWUSR | S_IXUSR))) == -1)
       {
         writeTerm("Cannot find file 'pgns.pls'");
         return;
       }

       if ((j = read(players_fd, read_buffer, BUFFER_SIZE)) == -1)
       {
         writeTerm("Cannot read from file 'pgns.pls'");
         return;
       }

       /* File empty */
       if (j == 0)
       {
         writeTerm("File 'pgns.pls' is empty (Type 'checkformat' to check how it is
formatted)");
         return;
       }
```

```
      /* Only 1 byte: file not formatted */
      if (j == 1)
       {
         writeTerm("File 'pgns.pls' not formatted (Type 'checkformat' to check how it is
formatted)");
         return;
       }

      /* 2 bytes but not formatted */
      if ((j == 2) &&
         (((read_buffer[0] != ' ') && (read_buffer[0] != '\n')) ||
          ((read_buffer[1] != ' ') && (read_buffer[1] != '\n'))))
       {
         writeTerm("File 'pgns.pls' not formatted (Type 'checkformat' to check how it is
formatted)");
         return;
       }

      /* Checks if file is formatted (i.e. if at the end of file "blank+blank"
         or "blank+newline" or "newline+blank" or "newline+newline" is found */
      j = 0;
      while ((read_buffer[j] != '\0') && (read_buffer[j] != EOF) && (j < BUFFER_SIZE))
       {
         if (((read_buffer[j] == ' ') || (read_buffer[j] == '\n'))  &&
             ((read_buffer[j + 1] == ' ') || (read_buffer[j + 1] == '\n')))
           break;

         j++;
       }

      if (j < BUFFER_SIZE)
       {
         /* end of file: file not formatted */
         if ((read_buffer[j] == '\0') || (read_buffer[j] == EOF))
           {
             writeTerm("File 'pgns.pls' not formatted (Type 'checkformat' to check how it
is formatted)");
             return;
           }
       }

      /* end of buffer: file not formatted */
      else
       {
         writeTerm("File 'pgns.pls' not formatted (Type 'checkformat' to check how it is
formatted)");
         return;
       }

      /* arguments not at the top of file: file not formatted */
      if ((read_buffer[0] == ' ') || (read_buffer[0] == '\n'))
       {
         writeTerm("File 'pgns.pls' not formatted or contains no arguments\n(Type
'checkformat' to check how it is formatted)");
         return;
       }

      j = 0;
      i = 0;
      while (TRUE)                          /* read the whole buffer */
       {
         /* copy arguments in 'start_args' array */
         while ((read_buffer[j] != ' ') && (read_buffer[j] != '\n'))
           {
             start_args[i] = read_buffer[j];
             i++;
             j++;
```

```
        }

      /* add blank between contiguous arguments */
      start_args[i] = ' ';
      i++;

      /* break if end of file */
      if (((read_buffer[j] == ' ') || (read_buffer[j] == '\n'))  &&
          ((read_buffer[j + 1] == ' ') || (read_buffer[j + 1] == '\n')))
        break;

      /* skip blank or newline between arguments */
      j++;

      if ((read_buffer[j] == ' ') || (read_buffer[j] == '\n'))
        break;
    }

    start_args[i - 1] = '\0';

    /* counts arguments */
    i = 0;
    while (start_args[i] != '\0')
    {
      if (start_args[i] == ' ')
        arg++;
      i++;
    }
    arg++;

    strcpy(handles, start_args);
    close(players_fd);
}

else                              /* input from keyboard */
  {
    ptr1 = str;

    /* skip 'start' string */
    while ((*ptr1 != ' ') && (*ptr1 != '\0'))
     ptr1++;

    /* takes place on the first argument */
    ptr2 = ptr1 + 1;
    i = 0;

    /* copies arguments */
    while ((*ptr2 != '\0') && (*ptr2 != '\n'))
    {
      start_args[i] = *ptr2;
      i++;
      ptr2++;
    }

    start_args[i] = '\0';

    /* counts arguments */
    i = 0;
    while (start_args[i] != '\0')
    {
      if (((start_args[i] == ' ') && (start_args[i + 1] != ' ')) ||
          (start_args[i + 1] == '\0'))
        arg++;
      i++;
    }
  }
```

```
  number_start_args = arg;

  /* check the number of arguments */
  if ((number_start_args == 0) && (start_input == KEYBOARD))
    {
      writeTerm("Missing arguments (Please type 'help' for commands synopsis)");
      return;
    }

  inserted_start_cmd = TRUE;       /* allows to break from the loop in main */
}



/*
 ------------------------------------------------------------
|                                                            |
| Called when any command is given. It controls what command |
| has been inserted and then carries out the appropriate     |
| actions.                                                   |
|                                                            |
 ------------------------------------------------------------
*/

void executeCmd(char *str)
{
  int input;
  char s[TYPE_SIZE];

  if (strncmp(str, "start", 5) == 0)                    /* start */
    return startCmd(str);

  if (strncmp(str, "input keyboard\n", 15) == 0)         /* input keyboard */
    {
      if (start_input == KEYBOARD)
       {
         printf("\nInput already set to be from keyboard\n\n");
         return;
       }

      start_input = KEYBOARD;
      printf("\nInput set to be from keyboard\n\n");
      return;
    }

  if (strncmp(str, "input file\n", 11) == 0)             /* input file */
    {
      if (start_input == PLAYERS_FILE)
       {
         printf("\nInput already set to be from file\n\n");
         return;
       }

      start_input = PLAYERS_FILE;
      printf("\nInput set to be from file\n\n");
      return;
    }

  if (strncmp(str, "log on\n", 7) == 0)                  /* log on */
    {
      if (log == ON)
       {
         printf("\nLog mode is already on\n\n");
         return;
       }

      log = ON;
```

```
    printf("\nLog mode on\n\n");
    return;
  }

if (strncmp(str, "log off\n", 8) == 0)                  /* log off */
  {
    if (log == OFF)
     {
       printf("\nLog mode is already off\n\n");
       return;
     }

    log = OFF;
    printf("\nLog mode off\n\n");
    return;
  }

if (strncmp(str, "log overwrite\n", 14) == 0)           /* log overwrite */
  {
    if (log_output == OVERWRITE)
     {
       printf("\nLog output already set to be overwritten\n\n");
       return;
     }

    log_output = OVERWRITE;
    printf("\nLog output set to be overwritten\n\n");
    return;
  }

if (strncmp(str, "log append\n", 11) == 0)              /* log append */
  {
    if (log_output == APPEND)
     {
       printf("\nLog output already set to be appended\n\n");
       return;
     }

    log_output = APPEND;
    printf("\nLog output set to be appended\n\n");
    return;
  }

if (strncmp(str, "interval\n", 9) == 0)                 /* interval */
  {
  insert1:
    printf("\nPlease insert interval time (mins): ");
    scanf("%d", &input);
    if (input < 0)
     {
       printf("Interval time cannot be a negative number\n");
       goto insert1;
     }

    interval = input * 60;
    printf("Interval time set to %d minutes\n\n", input);
    return;
  }

if (strncmp(str, "loop\n", 5) == 0)                    /* loop */
  {
  insert2:
    printf("\nPlease insert number of loops: ");
    scanf("%d", &input);
    if (input < 1)
     {
       printf("Number of loops must be at least 1\n");
```

```
      goto insert2;
    }

    loop = input;
    printf("Number of loops set to %d\n\n", input);
    return;
  }

if (strncmp(str, "loop infinite\n", 14) == 0)          /* loop infinite*/
  {
    loop = -1;
    printf("Number of loops set to 'infinite'\n\n");
    return;
  }

if (strncmp(str, "type\n", 5) == 0)                    /* type */
  {
    printf("\nPlease insert the type of game you want to save: ");
    scanf("%s", s);
    strcpy(type, s);

    /* types like '--' become ' --' */
    if (type[2] == '\0')
    {
      type[3] = '\0';
      type[2] = type[1];
      type[1] = type[0];
      type[0] = ' ';
    }

    /* types like '-' become '-  ' */
    if (type[1] == '\0')
    {
      type[3] = '\0';
      type[2] = type[1] = ' ';
    }

    printf("Type set to '%s'\n\n", type);
    return;
  }

if (strncmp(str, "ntry\n", 5) == 0)                     /* ntry */
  {
  insert3:
    printf("\nPlease insert number of connection tries: ");
    scanf("%d", &input);
    if (input < 1)
    {
      printf("Number of tries must be at least 1\n");
      goto insert3;
    }

    ntry = input;
    printf("Number of connection tries set to %d\n\n", input);
    return;
  }

if (strncmp(str, "remove on\n", 10) == 0)            /* remove on */
  {
    if (remotion == ON)
    {
      printf("\nHandles remotion is already on\n\n");
      return;
    }

    remotion = ON;
    printf("\nHandles remotion on\n\n");
```

```
      return;
    }

  if (strncmp(str, "remove off\n", 11) == 0)           /* remove off */
    {
      if (remotion == OFF)
       {
         printf("\nHandles remotion is already off\n\n");
         return;
       }

      remotion = OFF;
      printf("\nHandles remotion off\n\n");
      return;
    }

  if (strncmp(str, "checkformat\n", 12) == 0)          /* checkformat */
    {
      printf("\n");
      system("od -a pgns.pls");
      printf("\n");
      return;
    }

  if (strncmp(str, "default\n", 8) == 0)               /* default */
    {
      printf("\n - input mode: from %s", DEFAULT_INPUT == PLAYERS_FILE ? "file" :
"keyboard");
      printf("\n - log mode: %s", DEFAULT_LOG == ON ? "on" : "off");
      printf("\n - log output: %s", DEFAULT_LOG_OUTPUT == OVERWRITE ? "overwrite" :
"append");
      printf("\n - interval: %d minutes", DEFAULT_INTERVAL);
      if (DEFAULT_LOOP == INFINITE)
       printf("\n - loop: infinite");
      else
       printf("\n - loop: %d", DEFAULT_LOOP);
      printf("\n - type: %s", type);
      printf("\n - ntry: %d", DEFAULT_NTRY);
      printf("\n - handles remotion: %s\n\n", DEFAULT_REMOTION == ON ? "on" : "off");
      return;
    }

  if (strncmp(str, "exit\n", 5) == 0)                /* exit */
    exit(0);

  if (strncmp(str, "help\n", 5) == 0)                /* help */
    {
      printf("
 - start [USERS]: starts the program (type 'help start' to see more)
 - input MODE: sets input mode (type 'help input' to see more)
 - log OPTION: controls log mode (type 'help log' to see more)
 - interval: sets interval between connections (type 'help interval' to see more)
 - loop: sets number of connections (type 'help loop' to see more)
 - type: sets the type of games to save (type 'help types' to see more)
 - ntry: sets the number of connection tries (type 'help ntry' to see more)
 - remove MODE: wrong handles remotion control (type 'help remove' to see more)
 - checkformat: view of 'pgns.pls' format (type 'help checkformat' to see more)
 - default: print default values (type 'help default' to see more)
 - exit: exit from command shell (type 'help exit' to see more)

");
      return;
    }

  if (strncmp(str, "help start\n", 10) == 0)         /* help start */
    {
      printf("
```

```
 - start: starts the program, connects to ICC and saves games of the users passed as
arguments.
          Arguments may be passed by typing-in or from a file created by user.

          SYNOPSIS: start

                      Read arguments from file 'pgns.pls' (if input mode is set to be
from file,
                      default option)

                    start USER1 USER2 USER3 ...

                      Arguments are USER1 USER2 USER3 ... (if input mode is set to be
from keyboard)

");
      return;
    }

  if (strncmp(str, "help log\n", 8) == 0)           /* help log */
    {
      printf("
 - log: turns on (off) log mode; default is ON. If log mode is on the result of actions
        performed by the program is saved in a file named 'pgns.log'.
        It also sets log output to be overwritten every time the program is executed or
        to be appended at the end of 'pgns.log' file. Default is APPEND.

        SYNOPSIS: log on

                    Turn on log mode

                  log off

                    Turn off log mode

                  log overwrite

                    Overwrite log file

                  log append

                    Append output at the end of log file

");
      return;
    }

  if (strncmp(str, "help interval\n", 13) == 0)     /* help interval */
    {
      printf("
 - interval: sets the interval time for connection; default is 120 minutes. The program
             re-connects to server every N minutes. Value must be an integer not < 0.

             SYNOPSIS: interval

                         Value required after giving the command

");
      return;
    }

  if (strncmp(str, "help loop\n", 9) == 0)          /* help loop */
    {
      printf("
 - loop: sets the number of connections; default is 'infinite'. The program connects to
server,
```

```
         saves the games, disconnects, sleeps for the interval time and then reconnects;
this
         is repeated N times

         SYNOPSIS: loop

                     To insert an integer value for number of connections. Value must be
an
                     integer not < 1. Value required after giving the command

                   loop infinite

                     To set infinite number of connections

");
      return;
    }

  if (strncmp(str, "help type\n", 9) == 0)          /* help type */
    {
      printf("
 - type: sets the type of game to save (see README for types available on ICC).
Charachter '*'
         matches any character of type field; default is '16*' (rated or unrated
kriegspiel)

         SYNOPSIS: type

                     To insert new type. Value required after giving the command

");
      return;
    }

  if (strncmp(str, "help input\n", 10) == 0)        /* help input */
    {
      printf("
 - input: sets input mode to be from file or by typing-in. Default is from file.

          SYNOPSIS: input keyboard

                      To type-in arguments following 'start' command

                    input file

                      To read arguments from file 'pgns.pls' created by user

");
      return;
    }

  if (strncmp(str, "help ntry\n", 9) == 0)          /* help ntry */
      {
        printf("
 - ntry: sets the number of times the program tries to connect to server. If,
         after some failures, the program connects successfully then the
         program goes on; instead if it has tried NTRY times without connecting
         the program exits. Value must be an integer not < 1. Default is 5.

         SYNOPSIS: ntry

                     Sets connection tries. Value required after giving the command

");
        return;
      }
```

```
  if (strncmp(str, "help remove\n", 12) == 0)        /* help remove */
    {
      printf("
 - remove: turns on (off) wrong handles remotion; default is ON. If remotion is on
           when ICC says that the current handle doesn't match with any player's
           name then the handle is removed from file 'pgns.pls'. This is useful when
           an accout doesn't exists anymore.
           Handles remotion only works with input from file.

        SYNOPSIS: remove on

                    Turn on handles remotion

                  remove off

                    Turn off handles remotion

");
      return;
    }

  if (strncmp(str, "help checkformat\n", 17) == 0)  /* help checkformat */
    {
      printf("
 - checkformat: shows format of file 'pgns.pls' in order to check if it fulfills
                the required format. See README file to get information about the
                format required.

                SYNOPSIS: checkformat

                            View of 'pgns.pls' format

");
      return;
    }

  if (strncmp(str, "help default\n", 13) == 0)      /* help default */
    {
      printf("
 - default: shows default value for all commands available.

            SYNOPSIS: default

                        View of default values

");
      return;
    }

  if (strncmp(str, "help exit\n", 10) == 0)        /* help exit */
    {
      printf("
 - exit: exit from command shell. You can insert such command only at prompt, not
         after 'start' command was given.

            SYNOPSIS: exit

                        Exit from command shell

");
      return;
    }

  printf("\nUnknown command\n");
  printf("Please type 'help' for commands synopsis\n\n");
}
```

```
/*
 ------------------------------------------------------------
|                                                            |
| Read the next user from start_args, in order to send commands |
| to ICC (e.g. 'history user').                              |
|                                                            |
 ------------------------------------------------------------
*/

void getNextUser(int users)
{
  int i = 0;
  int number_of_users = number_start_args - users;   /* position of user in start_args */
  char *ptr = start_args;

  memset(user_string, '\0', strlen(user_string));

  /* skip users before the one we are looking for */
  while (number_of_users > 0)
    {
      if (*ptr == ' ')                               /* blanks are important! */
       number_of_users--;
      ptr++;
    }

  /* copy user in user_string */
  while ((*ptr != ' ') && (*ptr != '\0') && (*ptr != '\n'))
    {
      user_string[i] = *ptr;
      ptr++;
      i++;
    }

  user_string[i] = '\0';
}



/*
 ------------------------------------------------------------
|                                                            |
| Display prompt after which user can insert commands. You can  |
| change prompt form by changing the string 'char prompt'.   |
|                                                            |
 ------------------------------------------------------------
*/

char prompt = '#';

void displayPrompt(void)
{
  printf("%c", prompt);
  fflush(stdout);
}
```

```
/*
 ------------------------------------------------------------
|                                                            |
| This one is used to write messages in a log file. The service |
| is active only if log mode is on. Log file is named        |
| 'pgns.log' and it's created in main.                       |
| The form of a line of the log file is:                     |
|                                                            |
|    date_and_time:     message                              |
|                                                            |
| The argument 'char *str2' is used when a message with a    |
| string is written, e.g. 'cannot open file filename'. In    |
| other cases the function is called passing NULL.           |
|                                                            |
 ------------------------------------------------------------
*/

void writeLog(char *str1, char *str2)
{
  char *buffer;
  time_t t;

  if (log == ON)
    {
      chdir("..");                                /* exit from 'download' directory */

      t = time(NULL);
      buffer = ctime(&t);
      buffer[24] = '\0';
      write(fd_log, buffer, strlen(buffer));      /* writes date and time */
      buffer = ":     ";
      write(fd_log, buffer, strlen(buffer));      /* writes ':' and skips some blanks */
      buffer = str1;
      write(fd_log, buffer, strlen(buffer));      /* writes error message */
      if (str2 != NULL)
       {
         buffer = str2;
         write(fd_log, buffer, strlen(buffer));   /* writes string2 (if string2 is
                                                     passed) */
       }
      buffer = "\n";
      write(fd_log, buffer, strlen(buffer));      /* writes newline */

      chdir("download");                          /* go to 'download' directory */
    }
}



/*
 ------------------------------------------------------------
|                                                            |
| Print message on terminal if it is available.              |
|                                                            |
 ------------------------------------------------------------
*/

void writeTerm(char *str)
{
  if (term == ON)
    printf("\n%s\n\n", str);
}
```

```
/*
 ------------------------------------------------------------
|                                                            |
| This is used for important messages, such as errors. If log |
| mode is on it calls 'writeLog' function. If a term is      |
| available (i.e. user interacts with term and cron daemon is |
| not used) the message is printed on stdout.                |
|                                                            |
 ------------------------------------------------------------
*/

void writeLogAndTerm(char *str)
{
  writeLog(str, NULL);
  writeTerm(str);
}



/*
 ------------------------------------------------------------
|                                                            |
| This is called whenever ICC tells us that                  |
|                                                            |
|    "last_user_handle" does not match any player's name     |
|                                                            |
| The function looks for the handle in 'handles' array (if it |
| isn't found it does nothing). When it is found it is removed |
| from the array, then file 'pgns.pls' is overwritten with the |
| new list of handles and re-formatted.                      |
|                                                            |
 ------------------------------------------------------------
*/

void removeHandle(void)
{
  int i, j, start_string, found, length, fd;

  if (remotion == OFF)
    return;

  /* string search */
  found = FALSE;
  i = 0;
  while (!found && (handles[i] != '\0'))
    {
      j = 0;
      start_string = i;
      while (handles[i] != '\0')
       {
         if (handles[i] != user_string[j])
           {
             /* if it doesn't match go to the next handle */
             while ((handles[i] != ' ') && (handles[i] != '\n'))
              i++;
             i++;
             break;
           }
         else
           {
             /* if after last matching character we find end of array or handle-separator
                 the handle is found */
             if ((handles[i + 1] == ' ') || (handles[i + 1] == '\n') || (handles[i + 1] ==
'\0'))
              {
                found = TRUE;
                break;
```

```
            }

           i++;
           j++;
         }
      }
    }

  if (!found)
    return;

  /* fill the "hole" left by the removed handle */
  length = strlen(user_string) + 1;
  i = 0;
  while (handles[start_string + length + i] != '\0')
    {
      handles[start_string + i] = handles[start_string + length + i];
      i++;
    }

  /* fill with "\0" the exceeding part */
  while (handles[start_string + i] != '\0')
    {
      handles[start_string + i] = '\0';
      i++;
    }

  /* reaches the end of array replacing blanks with newlines */
  i = 0;
  while (handles[i] != '\0')
    {
      if (handles[i] == ' ')
       handles[i] = '\n';
      i++;
    }

  /* re-format the file: i.e. puts two newlines at the end */
  if ((handles[i - 1] != '\n') || (handles[i - 2] != '\n'))
    {
      handles[i] = handles[i + 1] = '\n';
      handles[i + 2] = '\0';
    }

  chdir("..");                                /* exit from 'download' directory */

  if ((fd = (open("pgns.pls", O_WRONLY | O_TRUNC))) == -1)
    {
      chdir("download");                      /* writeLog wants download directory */
      writeLogAndTerm("Cannot open file 'pgns.pls'");
      return;
    }

  nrem++;
  chdir("download");                          /* writeLog wants download directory */
  writeLog("Wrong handle removed from file 'pgns.pls': ", user_string);
  chdir("..");

  write(fd, handles, strlen(handles));
  close(fd);

  chdir("download");                          /* go to 'download' directory */
}
```

```
/* history_parser.h - Functions dealing with output of history command */



#ifndef HISTORY_PARSER
#define HISTORY_PARSER


#include "def.h"

#include <string.h>
#include <stdio.h>


char tmp_buffer[HISTORY_OUTPUT_SIZE];     /* where the output of history command is
                                             stored */
int tmpbuf_index;                         /* index for tmp_buffer array */
char filename[MAX_FILENAME_SIZE];         /* pointer to filename for new file */
char opponent[OPPONENT_SIZE];             /* where the opponent name is stored */
char data_and_time[DATA_AND_TIME_SIZE];   /* where the data and time are stored */
char number_of_game[NUMBER_OF_GAME_SIZE]; /* where the number of current game is
                                             stored */

extern char type[TYPE_SIZE];


void getNextGame(int ngame);

void getOpponent(void);

void getDataAndTime(void);

void getFirstGame(void);

int wantedType(void);

int abortedGame(void);

void concatenateToFilename(char *str);


#endif
```

```
/* history_parser.c - Functions dealing with output of history command */



#include "../h/history_parser.h"



/*
 ----------------------------------------------------------------
|                                                                |
| Skip one line. Looks for next newline and returns a pointer    |
| to the following charachter.                                   |
|                                                                |
 ----------------------------------------------------------------
*/

char *skipLine(char *start)
{
  char *ptr;

  ptr = start;
  while (*ptr != EOF)
    {
      if (*ptr == '\n')
       break;
      ptr++;
    }
  ptr++;

  return ptr;
}



/*
 ----------------------------------------------------------------
|                                                                |
| Reads from buffer where is stored the output of history        |
| command and sets the array 'number_of_game' to the first       |
| number of game of the buffer. The first means the 'higher      |
| line'. Remember that 'number_of_game' is an array of char,     |
| not an integer.                                                |
|                                                                |
 ----------------------------------------------------------------
*/

void getFirstGame(void)
{
  char *ptr;

  memset(number_of_game, '\0', strlen(number_of_game));

  /* starts at the top of the buffer and skip 4 lines */
  ptr = skipLine(tmp_buffer);
  ptr = skipLine(ptr);
  ptr = skipLine(ptr);
  ptr = skipLine(ptr);

  /* stores number of game (2 digits) and '\0' at the end */
  number_of_game[0] = *ptr;
  ptr++;
  number_of_game[1] = *ptr;
  number_of_game[2] = '\0';
}
```

```
/*
 ------------------------------------------------------------
|                                                            |
| Reads from buffer where is stored the output of history    |
| command and sets the array 'number_of_game' to the next    |
| number of game of the buffer. It starts from the current   |
| number of game and get the next one. The next means the 'line |
| below'. Remember that 'number_of_game' is an array of char,  |
| not an integer.                                            |
|                                                            |
 ------------------------------------------------------------
*/

void getNextGame(int ngame)
{
  char *ptr;

  if (ngame < (HISTORY_GAMES - 1))             /* if this is the last game do nothing */
    {
      ptr = tmp_buffer;
      while (TRUE)
       {
         /* looks in the buffer for current number of game */
         if (strncmp(ptr, number_of_game, 2) == 0)
           {
             /* go to the line below */
             ptr += HISTORY_LINE_LENGTH;
             number_of_game[0] = *ptr;
             ptr++;
             number_of_game[1] = *ptr;
             number_of_game[2] = '\0';
             break;
           }
         ptr = skipLine(ptr);
       }
    }
}



/*
 ------------------------------------------------------------
|                                                            |
| Reads from buffer where is stored the output of history    |
| command and sets the array 'opponent' to the name of the   |
| opponent for the current game (given by 'number_of_game').   |
|                                                            |
 ------------------------------------------------------------
*/

void getOpponent(void)
{
  char *ptr;
  int i;

  memset(opponent, '\0', strlen(opponent));

  ptr = tmp_buffer;
  while (TRUE)
    {
      /* looks in the buffer for current number of game */
      if (strncmp(ptr, number_of_game, 2) == 0)
       {
         /* skip characters to reach the opponent field */
         for (i = 1; i <= OPP_FIELD; i++)
           ptr++;
```

```
        i = 0;
        while (*ptr != ' ')
          {
            opponent[i] = *ptr;
            i++;
            ptr++;
          }

        opponent[i] = '\0';
        break;
       }
      ptr = skipLine(ptr);
    }
}



/*
 ----------------------------------------------------------------
|                                                                |
| Reads from buffer where is stored the output of history        |
| command and sets the array 'data_and_time' to the value of     |
| data and time for the current game (given by                   |
|'number_of_game').                                              |
|                                                                |
 ----------------------------------------------------------------
*/

void getDataAndTime(void)
{
  char *ptr;
  int i;

  memset(data_and_time, '\0', strlen(data_and_time));

  ptr = tmp_buffer;
  while (TRUE)
    {
      /* looks in the buffer for current number of game */
      if (strncmp(ptr, number_of_game, 2) == 0)
       {
         /* skip characters to reach the data and time field */
         for (i = 1; i <= DATA_FIELD; i++)
           ptr++;

         for (i = 0; i <= 2; i++)              /* month */
           {
             data_and_time[i] = *ptr;
             ptr++;
           }

         data_and_time[3] = '.';
         ptr++;

         for (i = 4; i <= 5; i++)              /* day */
           {
             data_and_time[i] = *ptr;
             ptr++;
           }

         data_and_time[6] = '.';
         ptr++;

         for (i = 7; i <= 8; i++)              /* year */
           {
             data_and_time[i] = *ptr;
             ptr++;
```

```
          }

        data_and_time[9] = '.';
        ptr++;

        for (i = 10; i <= 14; i++)            /* time */
          {
            data_and_time[i] = *ptr;
            ptr++;
          }

        data_and_time[15] = '\0';
        break;
       }
      ptr = skipLine(ptr);
    }
}



/*
 ----------------------------------------------------------------
|                                                                |
| Reads from buffer where is stored the output of history        |
| command and checks if the current game (given by               |
| 'number_of_game') is of correct type.                          |
|                                                                |
 ----------------------------------------------------------------
*/

int wantedType(void)
{
  char *ptr;
  char t[TYPE_SIZE];
  int i;

  ptr = tmp_buffer;
  while (TRUE)
    {
      /* looks in the buffer for current number of game */
      if (strncmp(ptr, number_of_game, 2) == 0)
       {
        /* skip characters to reach the type field */
        for (i = 1; i <= TYPE_FIELD; i++)
          ptr++;

        /* copy type from history buffer to t array */
        for (i = 0; i < (TYPE_SIZE - 1); i++)
          {
            t[i] = *ptr;
            ptr++;
          }

        t[i] = '\0';

        /* check if it is the wanted type */
        i = 0;
        while (type[i] != '\0')
          {
            if ((type[i] != t[i]) && (type[i] != '*'))
             return FALSE;
            i++;
          }

        return TRUE;
       }
      ptr = skipLine(ptr);
```

```
    }
}


/*
 ----------------------------------------------------------------
|                                                                |
| Reads from buffer where is stored the output of history        |
| command and checks if the current game (given by               |
| 'number_of_game') is aborted.                                  |
|                                                                |
 ----------------------------------------------------------------
*/

int abortedGame(void)
{
  char *ptr;
  char r;
  int i;

  ptr = tmp_buffer;
  while (TRUE)
    {
      /* looks in the buffer for current number of game */
      if (strncmp(ptr, number_of_game, 2) == 0)
       {
         /* skip characters to reach the result field */
         for (i = 1; i <= RESULT_FIELD; i++)
           ptr++;

         /* copy result from history buffer to r */
         r = *ptr;

         /* check if it is an aborted game */
         if (r == 'a')
           return TRUE;

         return FALSE;
       }
      ptr = skipLine(ptr);
    }
}


/*
 ----------------------------------------------------------------
|                                                                |
| Concatenate str to filename. Store the resulting string in     |
| filename array again.                                          |
|                                                                |
 ----------------------------------------------------------------
*/

void concatenateToFilename(char *str)
{
  int i;

  /* it put itself at the end of filename string */
  i = 0;
  while (filename[i] != '\0')
    i++;

  while (*str != '\0')
    {
      filename[i] = *str;
```

```
      str++;
      i++;
    }

  filename[i] = '\0';
}
```

```
/* iccfunct.h - Functions for input/output between ICC and the client */



#ifndef ICCFUNCT_H
#define ICCFUNCT_H


#include "def.h"
#include "stringfunct.h"
#include "history_parser.h"
#include "cmd.h"

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <ctype.h>



int fd;                          /* file descriptor to save games */

char return_word[WORD_SIZE + 1]; /* used in getWord function */
int server_desc;                 /* socket used to communicate with server */

int processing_history;          /* TRUE if we are parsing the output of history
                                    command */
int end_history_output;          /* TRUE if we have finished to parse the output of
                                    history command */
int processing_smoves;           /* TRUE if we are parsing the output of smoves
                                    command */
int end_smoves_output;           /* TRUE if we have finished to parse the output of
                                    smoves command */
int invalid_handle;              /* TRUE if server says that last handle is invalid */
string_list_t *buf_list;         /* buffer of strings to parse */
char buf[BUFFER_SIZE + 1];       /* buffer of ICC output that fills buf_list */
int nbuf;                        /* index for buf */
int just_got_escape;             /* it remembers if the previous character was a MARK */
int nesting_level;               /* nesting level of level1 datagram block */


void getToken(void);

void getQuotedString(int spec);

int getWord(void);

void sendToICC(char *str);

void login(void);

void level1Block(int depth);

int findOpen1(void);

void parseBuffer(string_list_t *buf_list);

void initParsing(void);

void parseChar(char c);

void parse(char *buffer, int n);


#endif
```

```
/* iccfunct.c - Functions for input/output between ICC and the client */



#include "../h/iccfunct.h"



/*
 ----------------------------------------------------------------
|                                                                |
| Send a string to ICC.                                          |
|                                                                |
 ----------------------------------------------------------------
*/

void sendToICC(char *str)
{
  int nwritten;
  int n;
  n = strlen(str);
  nwritten = write(server_desc, str, n);
  if (nwritten != n)
    writeLogAndTerm("EOF or I/O error to client");
}



/*
 ----------------------------------------------------------------
|                                                                |
| The following function gets the next character from the input  |
| stream. If the character is MARK it controls if the character  |
| after the MARK is a special character. If the character after  |
| the MARK is not one of the special ones, it just acts like     |
| the MARK was not there.                                        |
|                                                                |
 ----------------------------------------------------------------
*/

static int token;           /* 1 character read from server */
static int special;         /* special character; see within the routine */

void getToken(void)
{
  token = getOneChar();
  if (token == EOF)
    {
      writeLog("Input terminated unexpectedly", NULL);
      exit(1);
    }

  if (token != MARK)
    {
      special = FALSE;
      return;
    }

  token = getOneChar();
  special = (token == OPEN1 || token == CLOSE1 ||
             token == OPEN2 || token == CLOSE2 ||
             token == OPENQ || token == CLOSEQ);
}
```

```
/*
 ---------------------------------------------------------------
|                                                               |
| This function expects to find the remainder of a quoted       |
| string. By the time it gets here, we've already read a begin  |
| quote mark. It puts the result into the return_word variable. |
| It returns with us sitting on the quote mark. "spec" tells us |
| if we're requiring a special mark at the end, or just a       |
| CLOSEQ.                                                       |
|                                                               |
 ---------------------------------------------------------------
*/

void getQuotedString(int spec)
{
  int i;
  for (i = 0; i < WORD_SIZE; i++)
    {
      getToken();
      if (special == spec && token == CLOSEQ)
       break;
      return_word[i] = token;
    }

  return_word[i] = '\0';
  getToken();              /* skip the close quote */
}



/*
 ---------------------------------------------------------------
|                                                               |
| This function gets one datagram type string. The first        |
| character of this string has already been read. It skips      |
| preceeding blanks, and a blank (or CLOSE2 or any special      |
| mark) indicates the end of the string. It returns the result  |
| in return_word[]. If there is no word before the CLOSE2, it   |
| returns FALSE (leaving us sitting on the CLOSE2), otherwise   |
| returns TRUE. It leaves us sitting on the character AFTER the |
| last character read.                                          |
|                                                               |
 ---------------------------------------------------------------
*/

int getWord(void)
{
  int i;
  while ((!special) && isspace(token))
    getToken();

  if (special && token == CLOSE2)
    {
      return_word[0] = '\0';
      return FALSE;
    }

  if (token == OPENQ)
    {
      getQuotedString(special);
      return TRUE;
    }

  return_word[0] = token;
  for (i = 1; i < WORD_SIZE; i++)
    {
      getToken();
```

```
      if (special || isspace(token))
       break;
      return_word[i] = token;
    }

  return_word[i] = '\0';
  return TRUE;
}



/*
 ------------------------------------------------------------
|                                                            |
| Sends the login string. Here you can put code if you want to  |
| use an existing account on ICC. You can do that replacing     |
| "guest" with your login and adding 'sendToICC("password\n");' |
|                                                            |
 ------------------------------------------------------------
*/

void login(void)
{
  sendToICC("guest\n");
}



/*
 ------------------------------------------------------------
|                                                            |
| The following function is called to read in a level1 block.   |
| We just read an OPEN1 token to get here. This reads and        |
| process the rest of this block. getWord() is called twice to   |
| get the code number and the identity of the generator of       |
| this. This will work since the header of a level1 block        |
| always has these two parts, followed by some white space. The  |
| function leaves us sitting on the final CLOSE1.                |
|                                                            |
 ------------------------------------------------------------
*/

void level1Block(int depth)
{
  char command_number[20];                    /* number of command of the datagram */
  char player_name[20];                       /* name of player that receives the
                                                 datagram */
  char history[4] = "129", smoves[4] = "171";  /* code numbers for history and smoves
                                                 commands */
  char write_buffer;

  getToken();       /* getWord needs the first char of it to be read */
  getWord();
  safeStrcpy(command_number, return_word, sizeof(command_number));
  getWord();
  safeStrcpy(player_name, return_word, sizeof(player_name));

  for(;;)
    {
      if (special)
       {
         if (token == CLOSE1)
           {
             if (processing_history == TRUE)
              {
                processing_history = FALSE;
                end_history_output = TRUE;
```

```
            }
           if (processing_smoves == TRUE)
            {
              processing_smoves = FALSE;
              end_smoves_output = TRUE;
            }
           break;
         }
       else if (token == OPEN1)      /* start of level 1 datagram */
         level1Block(depth + 1);
     }

     /* read all characters but MARK and special characters */
     else if (token != '\r')           /* ignore all '\r's */
     {
       write_buffer = token;

       /* if the datagram refers to history command copy the output in tmp_buffer */
       if (strcmp(command_number, history) == 0)
         {
           /* if " is found it means that server says that last user handle were
               invalid, then it has to be removed from file 'pgns.pls' */
           if ((write_buffer == '"') && (start_input == PLAYERS_FILE))
            {
              /* if this is the second " found remove the handle */
              if (invalid_handle == TRUE)
                {
                  invalid_handle = FALSE;
                  removeHandle();
                }

              /* if invalid_handle is set to FALSE it means this is the first
                 " found, then do not remove user handle yet */
              else
                invalid_handle = TRUE;
            }

           /* copy history ouput */
           tmp_buffer[tmpbuf_index] = write_buffer;
           tmpbuf_index++;
           if (processing_history == FALSE)
            processing_history = TRUE;
         }

       /* if the datagram refers to smoves command copy the output in file fd */
       if (strcmp(command_number, smoves) == 0)
         {
           write(fd, &write_buffer, 1);
           if (processing_smoves == FALSE)
            processing_smoves = TRUE;
         }
     }
     getToken();
   }
}
```

```
/*
 ----------------------------------------------------------------
|                                                                |
| The following just gobbles input until it finds an OPEN1 type |
| token.                                                         |
|                                                                |
 ----------------------------------------------------------------
*/

int findOpen1(void)
{
  if (special && token == OPEN1)
    return TRUE;

  while (!(special && token == OPEN1))
    getToken();

  return TRUE;
}



/*
 ----------------------------------------------------------------
|                                                                |
| This one is called when we've found a level1 block to parse.  |
|                                                                |
 ----------------------------------------------------------------
*/

void parseBuffer(string_list_t *buf_list)
{
  setUpGetOneChar(buf_list);

  getToken();
  if (!findOpen1())
    {
      writeLog("Couldn't find OPEN1", NULL);
      exit(1);
    }

  level1Block(0);
}



/*
 ----------------------------------------------------------------
|                                                                |
| Initialize global variables used to parse the output of ICC.  |
|                                                                |
 ----------------------------------------------------------------
*/

void initParsing(void)
{
  buf_list = NULL;
  memset (buf, '\0', strlen(buf));
  nbuf = 0;
  just_got_escape = FALSE;
  nesting_level = 0;
}
```

```
/*
 ------------------------------------------------------------
|                                                            |
|  The char c has just been read from the ICC and it needs to  |
|  be parsed. If it's a complete level1 datagram, then this    |
|  function parses it and calls appropriate functions on the   |
|  various components of it.                                   |
|                                                            |
 ------------------------------------------------------------
*/

void parseChar(char c)
{
  int parse_block = FALSE;

  /* if buffer is full read and empty it */
  if (nbuf == BUFFER_SIZE)
    {
      buf[BUFFER_SIZE] = '\0';
      buf_list = putStringInList(buf_list, buf);
      nbuf = 0;
    }

  /* first put character c into the input buffer */
  buf[nbuf] = c;
  nbuf++;

  /* check if we read a MARK followed by OPEN1 or CLOSE1 */
  if (just_got_escape)
    {
      if (c == CLOSE1)
       {
         nesting_level--;
         parse_block = (nesting_level == 0);  /* parse block only when all nesting levels
                                                 are closed */
       }

      else if (c == OPEN1)
       nesting_level++;
    }

  just_got_escape = (c == MARK);

  if (parse_block)
    {
      /* a top-level level1 block has just been read, so parse it */
      buf[nbuf] = '\0';
      buf_list = putStringInList(buf_list, buf);
      buf_list = reverseSl(buf_list);
      parseBuffer(buf_list);

      freeStrings(buf_list);
      buf_list = NULL;
      nbuf = 0;
    }
}
```

```
/*
 ------------------------------------------------------------
|                                                            |
| Start to parse the buffer we've just read from socket stream. |
|                                                            |
 ------------------------------------------------------------
*/

void parse(char *buffer, int n)
{
  int i;
  for (i = 0; i < n; i++) parseChar(buffer[i]);
}
```

```
/* connection.h - Main function that connects to server and saves the games */



#ifndef CONNECTION_H
#define CONNECTION_H


#include "../h/iccfunct.h"
#include "../h/cmd.h"
#include "../h/history_parser.h"

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <ctype.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/time.h>
#include <errno.h>


extern int server_desc;                 /* socket used to communicate with server */
extern int saved_games;                 /* number of games saved */


void connection(void);


#endif
```

```
/* connection.c - Main function that connects to server and saves the games */



#include "../h/connection.h"



/*
 ------------------------------------------------------------
|                                                            |
| Here are the things it does:                               |
|                                                            |
|  - connect to ICC,                                         |
|  - for each user specified with 'start' command it:        |
|     * sends 'history user' command,                        |
|     * parses the output and store it in a buffer,          |
|     * it saves only game of type 'type',                   |
|     * it saves only new game, i.e. those games that haven't|
|       already caused a file creation,                      |
|     * it doesn't save aborted games,                       |
|     * for each game to save it:                            |
|        o sends 'smoves user game' command,                 |
|        o parses the output and save the game in a PGN file,|
|        o the filename is so-formed:                        |
|                                                            |
|           user-opponent.Month.day.year.h:m.pgn.            |
|                                                            |
 ------------------------------------------------------------
*/

char *main_server_address = "204.178.125.65";
int main_server_port = 5000;

void connection(void)
{
  struct sockaddr_in serv_addr;
  fd_set in_set, in_set_tmp;          /* used by select function */
  int nd;                             /* used by select function */
  int nread;                          /* used by read function */
  int users;                          /* number of users passed as argument of 'start' */
  int ngame;                          /* number of game of history table (1 to 20) */
  int nt;                             /* connection tries */
  char line_buffer[BUFFER_SIZE];      /* buffer used both for read from input and read
                                         from server */
  int sent_history_cmd;               /* TRUE if 'history' command was sent */
  int read_history_table;             /* TRUE if output of 'history' command was
                                         stored */
  int sent_smoves_cmd;                /* TRUE if 'smoves' command was sent */
  int sent_logout;                    /* TRUE if 'logout' was sent */
  char command_to_icc[ICC_CMD_SIZE];  /* commands more complex than a single string */

  users = number_start_args;
  ngame = 0;

  sent_history_cmd = read_history_table = sent_smoves_cmd = sent_logout = FALSE;
  processing_history = end_history_output = processing_smoves = end_smoves_output =
FALSE;

  bzero((char *)&serv_addr, sizeof(serv_addr));
  serv_addr.sin_family = AF_INET;
  serv_addr.sin_addr.s_addr = inet_addr(main_server_address);
  serv_addr.sin_port = htons(main_server_port);

  if ((server_desc = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    writeLogAndTerm("Cannot open stream socket");
```

```
  /* connection tries */
  nt = ntry;
  while (nt > 0)
    {
      if (connect(server_desc, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0)
       writeLog("Cannot connect to server", NULL);
      else
       {
         writeLog("Connected to server", NULL);
         break;
       }

      nt--;

      /* exit if tries are over */
      if (nt == 0)
       {
         writeLogAndTerm("Connection failed: exit\n");
         chdir("..");               /* exit from 'download' directory */
         rmdir("download");         /* remove it if empty */
         exit(1);
       }

      /* else wait and try again */
      sleep(1);
    }

  initParsing();

  login();
  sendToICC("set level1 1\n");       /* tells the server to turn on level1 datagram */

  FD_ZERO(&in_set);
  FD_SET(server_desc,&in_set);

  while (users >= 0)
    {
      memset(tmp_buffer, '\0', sizeof(tmp_buffer));
      tmpbuf_index = 0;

      sent_history_cmd = read_history_table = sent_smoves_cmd = FALSE;
      processing_history = end_history_output = processing_smoves = end_smoves_output =
FALSE;
      ngame = 0;

    listen:
      for(;;)
       {
         in_set_tmp = in_set;

         /* to break the listening when there is nothing to listen to */
         if (((sent_history_cmd && end_history_output && !sent_smoves_cmd) ||
              (end_smoves_output && sent_smoves_cmd) ||
              !sent_history_cmd) &&
             !sent_logout)
           {
             if (sent_smoves_cmd)
              {
                sent_smoves_cmd = FALSE;
                end_smoves_output = FALSE;
              }

             break;
           }

         if ((nd = select(server_desc + 1, &in_set_tmp, (fd_set *) 0, (fd_set *) 0, NULL))
< 0)
```

```
    {
      if (nd == -1 && errno == EINTR)
       continue;
      writeLog("Select error", NULL);
    }

  if (FD_ISSET(server_desc, &in_set_tmp))
    {
      nread = read(server_desc, line_buffer, sizeof(line_buffer));
      if (nread <= 0)
       break;

      /* parses what we have read */
      parse(line_buffer, nread);
    }
}

if ((!sent_history_cmd) && (users > 0))
{
  getNextUser(users);
  invalid_handle = FALSE;

  /* send 'history' command */
  sprintf(command_to_icc, "history %s\n", user_string);
  sendToICC(command_to_icc);
  sent_history_cmd = TRUE;
  goto listen;
}

if ((!read_history_table) && (users > 0))
{
  /* put EOF to avoid some problems */
  tmp_buffer[tmpbuf_index] = EOF;
  getFirstGame();
  read_history_table = TRUE;
}

if ((!sent_smoves_cmd) && (ngame < HISTORY_GAMES))
{
next_file:

  /* check game: if wrong type or aborted skip */
  if (!wantedType() || abortedGame())
    {
      getNextGame(ngame);

      if (ngame <= (HISTORY_GAMES - 2))
       {
         ngame++;
         goto next_file;
       }
      else
       goto next_user;
    }

  /* close file: does nothing first time or when file was not opened */
  close(fd);

  /* create filename */
  memset(filename, '\0', strlen(filename));
  strcpy(filename, user_string);
  concatenateToFilename("-");
  getOpponent();
  concatenateToFilename(opponent);
  concatenateToFilename(".");
  getDataAndTime();
  concatenateToFilename(data_and_time);
```

```
        concatenateToFilename(".pgn");

        /* open new file; if the file already exists nothing happens, nothing is saved
           in, just skip to next game to save. Thanks to this only new games are saved */
        if ((fd = open(filename, O_WRONLY | O_CREAT | O_EXCL, S_IRUSR | S_IWUSR |
S_IXUSR)) == -1)
          {
            getNextGame(ngame);
            if (ngame <= (HISTORY_GAMES - 2))
             {
               ngame++;
               goto next_file;
             }
            else
             goto next_user;
          }
        else
          {
            writeLog("Created file ", filename);
            saved_games++;
          }

        /* send 'smoves' command */
        sprintf(command_to_icc, "smoves %s %s\n", user_string, number_of_game);
        sendToICC(command_to_icc);
        sent_smoves_cmd = TRUE;

        getNextGame(ngame);
        ngame++;
        goto listen;
      }

    next_user:
      users--;
      if ((!sent_logout) && (users == 0))
       {
         /* send 'logout' */
         sendToICC("logout\n");
         sent_logout = TRUE;
         goto listen;
       }
    }

  writeLog("Disconnetted\n", NULL);
}
```

```
/* def.h - Constants and types definitions */



#ifndef DEF_H
#define DEF_H



/* list of strings */
typedef struct string_list_struct string_list_t;

struct string_list_struct
{
  string_list_t *next;
  char *string;
};


/* special characters */
#define MARK                        '\031'
#define OPEN1                          '['
#define CLOSE1                         ']'
#define OPEN2                          '('
#define CLOSE2                         ')'
#define OPENQ                          '{'
#define CLOSEQ                         '}'

/* boolean values */
#define FALSE                           0
#define TRUE                            1

/* mode */
#define OFF                             0
#define ON                              1

/* log output */
#define OVERWRITE                       0
#define APPEND                          1

/* loop connections */
#define INFINITE                       -1

/* input mode */
#define KEYBOARD                        0
#define PLAYERS_FILE                    1

/* default values */
#define DEFAULT_LOG                    ON
#define DEFAULT_LOG_OUTPUT         APPEND
#define DEFAULT_INTERVAL              120
#define DEFAULT_LOOP             INFINITE
#define DEFAULT_INPUT        PLAYERS_FILE
#define DEFAULT_NTRY                    5
#define DEFAULT_REMOTION               ON

/* size of arrays */
#define MAX_FILENAME_SIZE              60
#define ICC_CMD_SIZE                   30
#define WORD_SIZE                    2048
#define BUFFER_SIZE                  1000
#define HISTORY_OUTPUT_SIZE          2000
#define OPPONENT_SIZE                  13
#define DATA_AND_TIME_SIZE             16
#define NUMBER_OF_GAME_SIZE             3
#define USER_STRING_SIZE               16
```

```
#define TYPE_SIZE                     4
#define START_ARGS_SIZE            1000
#define PLAYERS_BUFFER_SIZE        1000

/* constants related to output of 'history' command, stored in tmp_buffer */
#define HISTORY_GAMES                20
#define HISTORY_LINE_LENGTH          68
#define OPP_FIELD                    18
#define DATA_FIELD                   52
#define TYPE_FIELD                   32
#define RESULT_FIELD                  4


#endif
```

```
/* main.c - Main program */



#include "../h/iccfunct.h"
#include "../h/cmd.h"
#include "../h/history_parser.h"
#include "../h/connection.h"

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <ctype.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/time.h>
#include <errno.h>
#include <signal.h>



void sig_int(int);                    /* interrupt handler, defined below */



/*
 ------------------------------------------------------------
|                                                            |
| This is the main program. It allows user to insert commands  |
| and when 'start' command is given it creates log file if log |
| mode is on and download directory if it doesn't already      |
| exist; then it repeatedly calls the function 'connection',   |
| according to the value of 'loop' variable.                   |
| User can force the exit from program by sending an interrupt |
| CTRL+C (see handle 'sig_int').                               |
| Main function can be also called passing 'start' as command  |
| line argument, then the program starts immediatly without    |
| waiting for user to insert commands. This is useful when you |
| use PGN-Saver with cron daemon.                              |
| Below you see the 'type' array with the default value of type |
| of game to save. Change it if you like (see README file or   |
| type 'help type' at the prompt of the program for details).  |
|                                                            |
 ------------------------------------------------------------
*/

char type[TYPE_SIZE] = "16*";         /* type of game to save */
int saved_games;                      /* number of saved games */

int main(int argc, char *argv[])
{
  char cmd_buffer[BUFFER_SIZE];       /* buffer used to store user command */
  char log_str[60];                   /* used to pass complex strings to writeLogAndTerm
                                         function */
  int log_output_flag;                /* overwrite or append */

  log = DEFAULT_LOG;                  /* log mode (see def.h) */
  log_output = DEFAULT_LOG_OUTPUT;    /* treatment of log output (see def.h) */
  interval = DEFAULT_INTERVAL * 60;   /* interval time (see def.h) */
  loop = DEFAULT_LOOP;                /* loop connections (see def.h) */
  start_input = DEFAULT_INPUT;        /* for reading input arguments (see def.h) */
```

```
  ntry = DEFAULT_NTRY;                /* connection tries (see def.h) */
  remotion = DEFAULT_REMOTION;        /* remove wrong handle from 'pgns.pls' */

  saved_games = 0;
  inserted_start_cmd = FALSE;
  nrem = 0;
  term = OFF;

  /* main called without command-line arguments */
  if (argc == 1)
    {
      term = ON;

      writeTerm("
************************
   Welcome to PGN-Saver
************************\n\n\n");

      /* command input */
      while (TRUE)
       {
         displayPrompt();
         memset(cmd_buffer, '\0', strlen(cmd_buffer));
         read(STDIN_FILENO, cmd_buffer, sizeof(cmd_buffer));
         executeCmd(cmd_buffer);

         if (inserted_start_cmd)
           break;
       }
    }

  /* if main is called passing command-line arguments log mode must be on and input
     mode must be set to be from file */
  if (argc > 1)
    {
      log = ON;
      log_output = APPEND;
      start_input = PLAYERS_FILE;
    }

  /* create log file if log mode is on */
  if (log == ON)
    {
      if (log_output == OVERWRITE)
       log_output_flag = O_TRUNC;
      else
       log_output_flag = O_APPEND;

      if ((fd_log = open("pgns.log", O_RDWR | O_CREAT | log_output_flag, S_IRUSR |
S_IWUSR | S_IXUSR)) == -1)
       writeTerm("Cannot create log file\n");
    }

  /* main called with command-line arguments: error handling */
  if ((argc > 1) &&
      ((strcmp(argv[1], "start") != 0) ||                /* incorrect 'start' command */
       (argc > 2)))                                      /* too many arguments */
    {
      writeLog("Incorrect command line\n", NULL);
      exit(1);
    }

  /* main called with 'start' as command-line argument */
  if ((argc == 2) && (strcmp(argv[1], "start") == 0))
    executeCmd("start");
```

```
  /* create download directory; if it already exists nothing happens */
  mkdir("download", S_IXUSR | S_IWUSR | S_IRUSR);
  chdir("download");

  writeLog("-------------- NEW SESSION --------------\n", NULL);

  if (signal(SIGINT, sig_int) == SIG_ERR)
    writeLog("Cannot catch SIGINT", NULL);

 start:
  connection();

  if (loop == INFINITE)              /* loop = infinite */
    {
      sleep(interval);
      goto start;
    }
  else                               /* loop = value */
    {
      loop--;
      if (loop > 0)
       {
         sleep(interval);
         goto start;
       }
    }

  chdir("..");                       /* exit from 'download' directory */

  /* try to remove 'download' directory. It is removed only if it is empty */
  rmdir("download");

  sprintf(log_str, "%d game%s saved\n", saved_games, saved_games != 1 ? "s" : "");
  writeLogAndTerm(log_str);

  /* print if any handles was removed */
  if (nrem > 0)
    {
      sprintf(log_str, "WARNING: %d wrong handle%s removed from file 'pgns.pls'\n", nrem,
nrem != 1 ? "s" : "");
      writeLogAndTerm(log_str);
    }

  return 0;
}



/*
 ------------------------------------------------------------
|                                                            |
| Handle for interrupt sent by user (CTRL+C). It exits from  |
| program and prints the number of games saved. Anyway you   |
| had better use such command only when program is sleeping, |
| in order to avoid to cause problems with file saving.      |
|                                                            |
 ------------------------------------------------------------
*/

void sig_int(int signo)
{
  char log_str[60];                    /* used to pass complex strings to
                                          writeLogAdnTerm function */

  if (signo == SIGINT)                 /* CTRL+C */
    {
      chdir("..");                     /* exit from 'download' directory */
```

```
      /* try to remove 'download' directory. It is removed only if it is empty */
      rmdir("download");

      sprintf(log_str, "%d game%s saved\n", saved_games, saved_games != 1 ? "s" : "");
      writeLogAndTerm(log_str);

      /* print if any handles was removed */
      if (nrem > 0)
       {
         sprintf(log_str, "WARNING: %d wrong handle%s removed from file 'pgns.pls'\n",
nrem, nrem != 1 ? "s" : "");
         writeLogAndTerm(log_str);
       }

      writeLogAndTerm("Interrupt sent by user: exit\n");
      exit(0);
    }
}
```

Bibliografia e riferimenti

Ciancarini P., *Giocatori Artificiali*, Milano, Mursia, 1992

Santi A., *Un programma a conoscenza distribuita per il gioco degli scacchi*, Tesi di laurea, Università degli Studi di Bologna, 1993.

http://www.galahadnet.com/chess/chessprg/bitboard.htm
Una guida alla struttura dati bitboard.

Lentini F., "L'altra intelligenza", 1997, http://web.tin.it/eloisa/ia.htm

http://www.tim-mann.org/chess.html
Home page di Tim Mann.

Chess Programming Theory, http://www.ast.cam.ac.uk/~cmf/chess/theory.html

Chess Tree Search, http://www.xs4all.nl/~verhelst/chess/search.html

"Deep Blue Technology", http://www.research.ibm.com/know/blue.html

http://www.cis.uab.edu/info/faculty/hyatt/hyatt.html
Home page di Robert Hyatt.

“How Deep Blue Works”,
http://www.research.ibm.com/deepblue/meet/html/d.3.2.html

Drew McDermott, “How intelligent is Deep Blue?”,
http://www.nyu.edu/gsas/dept/philo/courses/mindsandmachines/Papers/mcdermott.html

http://www.research.ibm.com/deepblue/learn/html/index.html
Dal sito dell’IBM. Il progetto Deep Blue: la sfida con Kasparov, la tecnologia.

Eppstein D., “Board rapresentations”, 1997,
http://www1.ics.uci.edu/~eppstein/180a/970408.html

Eppstein D., “Which nodes to search? Full-width vs. selective search”, 1999,
http://www1.ics.uci.edu/~eppstein/180a/990204.html

“Informatica scacchistica: la storia”,
http://www.maskeret.com/italiascacchistica/a_inform.htm

Grimaldi F., “Programmare il gioco degli scacchi”, 2001,
http://www.itportal.it/developer/algoritmi/soluzioni_43/

Grimaldi F., “Scacchi, una struttura dati”, 2001,
http://www.itportal.it/developer/algoritmi/soluzioni_44/

http://it.geocities.com/pietro_valocchi/lamosca/index.html
Dal sito di LaMoSca: come programmare un gioco per scacchi.

http://www.seanet.com/~brucemo/index.htm
Un sito dedicato alla programmazione scacchistica.

http://scacchi.qnet.it/manuale/comp08.htm
Sito dedicato a scacchi e computer.