UNIVERSITÀ DEGLI STUDI DI MILANO
Facoltà di Scienze Matematiche, Fisiche e Naturali
Corso di Laurea in Informatica

"Progettazione e sviluppo di un software a supporto dell'analisi di problemi di scacchi diretti in due mosse"

RELATORE:
Giovanni Righini

CORRELATORE:
Ing. Marco Guida

TESI DI:
Gianluca Passeri

MATRICOLA:
641544

Anno Accademico 2004/05

Nel loro angolo austero, i giocatori
dirigono i lenti pezzi. La scacchiera
li incatena alla sua severa
dimensione in cui battagliano due colori.
In essa, le figure incutono religiosi timori
la regina armigera, l'omerica torre,
l'agile cavallo, lo scortato re,
l'obliquo alfiere e i pedoni invasori.
Quando i giocatori se ne saranno andati,
quando anche il tempo li avrà consumati,
continuerà ancora ad officiarsi il rito.
In Oriente scoppiò questa guerra
il cui teatro è oggi tutta la terra.
Come l'altro, questo gioco è infinito.

Jorge Luis Borges

INTRODUZIONE pag. 5
1. MOTIVAZIONI
2. BREVE RIASSUNTO
2.1 SCOPO
2.2 MODALITA' OPERATIVE
2.3 RISULTATI

PROBLEMI SCACCHISTICI pag. 9
1. INTRODUZIONE
1.1 QUADRO GENERALE
1.2 COSA SONO
1.3 I PROBLEMI OGGI
2. SCACCO DIRETTO #2
2.1 UN PO' DI STORIA
2.2 CARATTERISTICHE
2.3 IL COMPOSITORE
2.4 IL SOLUTORE
2.5 MATTO #2 NEL DETTAGLIO
2.5.1 LA CHIAVE
2.5.2 LA MINACCIA
2.5.3 LE DIFESE
2.5.4 IL MATTO
2.5.5 ESEMPI
2.5.5.1 ESEMPIO 1 - MATTO IN 2
2.5.5.2 ESEMPIO 2 - BLOCCO
2.5.5.3 ESEMPIO 3 - NESSUNA SOLUZIONE
2.5.5.4 ESEMPIO 4 - PIU' SOLUZIONI
2.5.5.5 ESEMPIO RIASSUNTIVO
2.6 PROBLEMI E QUALITA'

IL PROGRAMMA pag. 24

1. INTRODUZIONE
2. IL PROGRAMMA
    - 2.1 SCHEMA
        - 2.1.1 DESCRIZIONE
    - 2.2 IL FILE
        - 2.2.1 FEN
    - 2.3 LETTORE FEN
        - 2.3.1 IL CODICE
    - 2.4 LA SCELTA DEL MOTORE
    - 2.5 TSCP
        - 2.5.1 BREVE DESCRIZIONE
        - 2.5.2 DETTAGLI TSCP
        - 2.5.3 TSCP E MATTO #2
    - 2.6 L'ALBERO DI RICERCA
        - 2.6.1 I LIVELLI DELL'ALBERO
        - 2.6.2 AND E OR
        - 2.6.3 QUALCHE NUMERO
        - 2.6.4 TAGLI
    - 2.7 L'ALGORITMO
        - 2.7.1 SEARCH
    - 2.8 MINACCIA E BLOCCO
        - 2.8.1 MINACCIA
            - 2.8.1.1 IL CODICE
        - 2.8.2 BLOCCO
    - 2.9 DIFESE EFFICACI
    - 2.10 IDENTIFICATORE DI PATTERN
        - 2.10.1 MATTO A STELLA
            - 2.10.1.1 IL CODICE
        - 2.10.2 MATTO A CROCE
        - 2.10.3 ROSA DI CAVALLO NERO
    - 2.11 CENNO AGLI OUTPUT

CONCLUSIONI pag. 70

1. RISULTATI OTTENUTI
2. ALTRI ELEMENTI DI ANALISI
3. ELEMENTI UTILI ALLA COMPOSIZIONE
4. ALTRE UTILITA'
5. SVILUPPI FUTURI

introduzione

1. MOTIVAZIONI

Desidero innanzitutto motivare la scelta di questo argomento per la mia tesi, prima di entrare nel dettaglio e presentare il lavoro compiuto negli ultimi sei mesi.
L'idea di imbattermi nell'ambito informatico-scacchistico nasce dall'unione di due materie che trovo particolarmente interessanti e sono proprio l'informatica e gli scacchi. Mentre la prima è una disciplina che mi coinvolge da anni ormai, gli scacchi sono riusciti ad affascinarmi da un breve periodo a questa parte spingendomi alla ricerca di qualcosa di ben più profondo di una semplice partita fra amici. Il gioco degli scacchi non è solo un passatempo per me: è molto di più. Ritengo che il Nobil Giuoco sia uno sport a tutti gli effetti. Il gioco degli scacchi inoltre, sviluppa il senso critico e la capacità di analisi, insegna a valutare tutte le alternative ed abitua al rispetto delle regole. C'è chi l'ha definito: «una palestra per la mente» (Blaise Pascal), chi «un grande spreco di cervelli» (lo scrittore sir Walter Scott) e chi «uno sport violento» (Marcel Duchamp). Ma forse è stato il campione Boris Spassky ad aver pronunciato la sentenza finale: «Il gioco degli scacchi è come la vita».
Giocare permette la creazione di propri schemi e vederli realizzare provoca piacere sia competitivo che intellettuale. Sono pienamente d'accordo con Siegbert Tarrasch quando dice: «...il gioco degli Scacchi, non è solamente il più nobile e il più bello di tutti i giochi, ma procura anche i più grandi piaceri intellettuali, perchè si colloca alla frontiera fra il gioco, l'arte e la scienza».
Solo ora che ho portato a termine la mia tesi, inizio a rendermi conto di quanto questo mondo sia sconfinato, ma allo stesso tempo affascinante e di quanto studio sia richiesto per riuscire a comprenderlo. Una cosa è certa: non voglio fermarmi di fronte ai suoi seducenti richiami.
D'altro canto, l'informatica è il principale dei miei interessi. Sono sempre rimasto a bocca aperta di fronte a tutto ciò che riguarda la tecnologia e in particolare dall'informatica che, mai come ai nostri giorni, spicca in qualsiasi ambito.
Per questi motivi ho deciso di intraprendere studi tecnico-scientifici, iniziando da un istituto tecnico con indirizzo informatico dove mi sono diplomato nel 2002, per poi proseguire la mia formazione presso il Polo di Crema frequentando un Corso di Laurea in Informatica. Durante tutti questi anni ho conosciuto i diversi aspetti legati a questa scienza e a tutti i suoi impieghi, ma l'ambito che più mi sta a cuore è quello connesso alla programmazione. Adoro programmare e voglio fare di questa passione una professione.
Terminati gli esami previsti per il corso di laurea, ho iniziato la ricerca di una tesi che mi permettesse di concludere il mio percorso di studi. Fra le tante attività proposte dai docenti, ho preferito orientarmi verso un campo pratico che fosse in grado di conciliare i miei interessi. Non appena mi fu presentata la possibilità di sviluppare un software a beneficio dei compositori di problemi scacchistici, vidi concretizzarsi l'unione di due grandi interessi: l'informatica e gli scacchi.
Dopo aver letto attentamente tutte le richieste, mi resi conto che quello era proprio uno dei settori che mi sarebbe piaciuto approfondire e da subito iniziai i primi lavori.

2. BREVE RIASSUNTO

2.1 Lo scopo

Scopo di questa tesi è stato quello di sviluppare un software a supporto dei compositori di problemi scacchistici che permetta loro un'automatica verifica della correttezza delle posizioni, in modo da potersi concentrare maggiormente sugli aspetti creativi del problema. Il programma tende a rispondere a tutte le esigenze fondamentali del compositore, prima calcolando tutte le soluzioni del problema e poi delineando i principali aspetti che lo caratterizzano. Oltre ad esaminare l'ammissibilità, il software deve quindi riuscire a dare un quadro generale della composizione ideata dall'utente.

2.2 Modalità operative

Dopo aver definito i requisiti dell'applicativo, si è passati ad uno stadio di analisi da cui sono state tratte le modalità operative necessarie per portare avanti il lavoro. Il processo che mi ha condotto al completamento del software, è composto da diverse fasi che hanno man mano contribuito ad un'evoluzione mirata ad estenderne le funzionalità.
Prima di tutto, si è pensato di fondare le basi su cui costruire il programma, ovvero la definizione delle regole che determinano i movimenti dei pezzi e di quelle che gestiscono l'avvicendamento dei turni. Per minimizzare i tempi, ho riutilizzato codice già esistente in altri applicativi impiegati per la gestione di competizioni di gioco attivo. Sarebbe stato uno spreco di tempo creare dal nulla funzioni già note.
Una seconda fase ha richiesto la stesura dell'algoritmo di ricerca delle soluzioni che oltre ad osservare le norme tipiche del gioco degli scacchi, deve anche far fronte ai canoni costruttivi imposti nel mondo della problemistica.
La terza fase sfrutta la precedente per estrarre tutti gli elementi in grado di caratterizzare il problema. E' proprio qui che avviene la distinzione fra problemi a blocco e problemi a minaccia, oppure la selezione di difese efficaci.
Un'ultima fase si occupa di riunire tutte le informazioni raccolte per stabilire l'esistenza di alcuni pattern noti ai compositori.
Si deve infine aggiungere una fase molto importante per la determinazione della qualità di un software, ovvero quella costituita dalle numerose funzioni di contorno che hanno permesso un utilizzo più flessibile dell'intero applicativo e una presentazione intelligente degli output generati dal programma.

2.3 Risultati

Per determinare l'efficacia dell'applicativo sono stati effettuati numerosi test, sia a lavoro ultimato sia in fase di sviluppo. I campioni utilizzati per verificare la correttezza del programma sono stati assortiti il più possibile in modo da ottenere una maggiore copertura dei casi che si possono incontrare. Un testing esaustivo, d'altra parte, sarebbe stato impossibile, poiché avrebbe richiesto molto tempo e la ricerca di problemi particolari con cui mettere alla prova la robustezza del programma.
I risultati ottenuti sono stati confrontati sia con una specifica bibliografia che con un software analogo già diffuso nel mondo dei compositori ed in entrambi i casi si sono rivelati corretti.

Oltre alla dimostrazione della validità del programma, si deve tener presente anche della rapidità con cui le soluzioni e tutti gli altri elementi vengono calcolati. Nel caso del programma in questione, la ricerca non si è mai spinta oltre il decimo di secondo su macchine di recente tecnologia.

problemi scacchistici

1. INTRODUZIONE

1.1 Quadro generale

Prima di passare alla presentazione del programma ritengo indispensabile fare una panoramica generale di quello che è la problemistica oggi. D'altra parte è fondamentale comprendere il lavoro che l'utilizzatore del programma deve compiere se gli si vuole fornire un software che sappia rispondere alle sue esigenze. La presentazione della problemistica dovrebbe inoltre condurre il lettore alla comprensione delle scelte implementative impiegate per portare a termine il lavoro.

1.2 Cosa sono

Un **problema** è una posizione artificiale (nel senso che ben difficilmente può essere raggiunta in una partita vera) che richiede di realizzare un enunciato in un dato numero di mosse [1].
In un problema di scacchi la possibilità di concretizzare l'enunciato nel numero di mosse indicato è nascosta e la dimostrazione di tale possibilità rappresenta la **soluzione** [2].
Appare evidente che la sfida non è più fra due giocatori, bensì tra compositore e risolutore.
In un problema non ha alcuna importanza se un "colore" ha un vantaggio materiale notevole. Lo scopo non è quello di vincere la partita, ma di trovare una soluzione ad un quesito scacchistico: l'enunciato.
A dimostrazione che in un problema, la superiorità materiale può essere ininfluente, propongo la seguente situazione:

Figura 1: Il bianco ha mattato

1.3 I problemi oggi

Il problema di scacchi costituisce una disciplina scacchistica con caratteristiche molto diverse dal consueto “gioco attivo”, a tal punto che, all'interno della FIDE (Fédération Internazionale des Échecs, cioè l'ente che si occupa di gestire competizioni scacchistiche a livello internazionale e continentale) [3] è stata da tempo costituita una specifica commissione che se ne occupa [4]. Sotto l'ampio cappello del Problema di Scacchi, si identificano poi una grande varietà di tipologie di problemi.
I tipi di problema più comuni sono:

- Matti diretti
- Aiutomatti
- Automatti
- Matti reflex
- Serie di mosse
- Retroanalisi
- Partite d'esempio

È poi possibile variare le regole e le caratteristiche dei pezzi e della scacchiera. Si entra in tal caso nel mondo degli scacchi eterodossi, oggi molto popolare fra i compositori.
L'oggetto della tesi non ricopre tutte queste varianti del gioco degli scacchi, ma solo ed esclusivamente i problemi di Matti diretti in due mosse (#2).

2. LO SCACCO MATTO #2

2.1 Un pò di storia

Sebbene vecchio di centinaia d'anni, il problema di scacchi #2 ha visto una prima grande evoluzione a partire dalla metà dell'800, sulla scia di alcuni gruppi di compositori, tra cui numerosi italiani, che si sono concentrati sulla realizzazione di problemi di matto in due mosse che illustrano e sfruttano in modo elementare i meccanismi di base del gioco (**elementi tematici**), quali ad esempio interferenze tra pezzi dello stesso colore o diverso; inchiodature, controscacchi, autoblocchi.
Con il trascorrere del tempo, altri compositori si sono sbizzarriti nel combinare elementi tematici di base per ottenere meccanismi più complessi (o **temi**), oggi noti con nomi dei compositori che per primi li hanno realizzati(Grimshaw, Novotny, …) o con nomi generici laddove la paternità è incerta.

2.2 Caratteristiche

Il problema di scacco matto diretto in 2 mosse appartiene alla variante del gioco degli scacchi dei matti diretti, ossia quell'insieme di problemi in cui il bianco matta il nero nel numero di mosse prefissato, mentre il nero contrappone le sue mosse migliori.

Vi sono regole del gioco attivo che perdono valore quando si passa al problema dello scacco matto #2. Ad esempio:

1. Non esiste la regola delle 50 mosse
2. Non esiste la patta per la triplice ripetizione

A differenza del più comune e conosciuto "gioco attivo", il problema dello scacco matto diretto in 2 mosse gode di caratteristiche particolari. Infatti:

1. Ogni problema ha una configurazione iniziale di pezzi differente
2. Non tutti i pezzi devono necessariamente apparire sulla scacchiera (ad eccezione dei Re)
3. I pezzi, Re escluso, possono presentarsi in più copie, come se provenissero da promozioni di pedoni durante il gioco attivo.
   Quindi non potranno esistere più di 9 Regine o più di 8 pedoni.
   Tuttavia, convenzioni adottate dalla comunità dei compositori tendono ad evitare l'esistenza di pezzi provenienti da promozioni.
   Due Alfieri sulle diagonali dello stesso colore non sono ammessi.
4. I pezzi possono apparire in qualunque posizione purché consentita e i Re non devono essere minacciati. Vi sono altre situazioni che non sono consentite, poiché è impossibile che derivino da una comune partita.
   Eccone alcune [2]:

Figura 2: Posizioni illegali

Ci sono ben 4 pezzi che non potrebbero trovarsi nella loro attuale posizione se provenissero da una partita qualunque, ossia:

- Ah8: Per raggiungere quella casa, l'Alfiere avrebbe dovuto passare per g7, ma è occupata da un pedone nero. Poiché i pedoni non hanno facoltà di retrocedere, se ne deduce che il pedone sia sempre rimasto in quella casa impedendo il passaggio dell'Alfiere
- Rh2: Due vie potrebbero portare il Re nero in quella casa. La prima richiede il passaggio per la settima o per l'ottava colonna, ma è impossibile perché il Re verrebbe minacciato dai pedoni in seconda traversa. La seconda richiede il passaggio del Re per la quarta colonna, ma in particolare per la casa f1, dove da sempre giace un Alfiere bianco
- Ra1: Rispetto alla torre in b2. Se la torre ha raggiunto b2 significa che è passata necessariamente per la prima traversa e quindi per la casa b1, imponendo l'ovvia minaccia al Re che avrebbe dovuto sicuramente spostarsi su un'altra traversa.
- Pa4: Gli unici modi che hanno i pedoni bianchi per raggiungere a4 sono o con l'avanzata del pedone in a2 o con una o più catture dei pedoni in b2 e c2. La presenza dei pedoni in a2, b3 e c2 rendono illegale il pedone in a4.

2.3 Il compositore

Il compositore ha pieni poteri sul problema: può gestire sia i pezzi bianchi che quelli neri e può decidere dove posizionarli per raggiungere il suo scopo, nonché caratterizzare il problema attraverso l'impiego di elementi tematici.
Il compositore cerca di creare qualcosa di elegante e piacevole che possa invitare l'attenzione di chi si lascia incuriosire da un problema. E' proprio lui che lancia una sfida al risolutore proponendo uno schema avvincente, ma allo stesso tempo ricco di tranelli e ostilità.

2.4 Il solutore

Il compito del solutore non è soltanto quello di dimostrare la propria abilità nel trovare la soluzione, ma è anche quello di individuare gli elementi che stanno alla base della combinazione vincente, evidenziando così il significato che il compositore ha voluto attribuire alla posizione ed al movimento dei pezzi nella soluzione. E' proprio così che nascono i concetti di tema ed elementi tematici. Ovviamente il solutore può andare oltre nell'analisi della composizione sino ad individuare i ruoli dei singoli pezzi, ricostruendo così in modo completo il percorso creativo del compositore. E' utile anche il lavoro di sintesi che nel proporre i meccanismi di gioco dà significato alla creazione. Detto ciò è facilmente comprensibile il paragone tra il problema di scacchi e l'opera artistica.

2.5 Matto #2 nel dettaglio

Il problema dello scacco matto diretto in 2 mosse è la variante più conosciuta fra chi non è particolarmente appassionato di scacchi. Questa tipologia di problema appare perfino sulle riviste enigmistiche. Quindi, chiunque conosca le regole del Nobil Giuoco potrebbe cimentarsi in questa modalità di gioco, magari iniziando dal lato del risolutore, poiché quello del compositore richiede competenze più elevate, soprattutto dal punto di vista teorico.
Ora però entriamo nel dettaglio del problema #2 e cerchiamo di capire come funziona.

Qualsiasi problema di scacco matto #2 segue un preciso schema identificabile in 4 elementi. La soluzione del problema è formata da:

1. **Chiave**
2. **Minaccia**
3. **Difesa**
4. **Matto**

Vediamo ora uno ad uno i singoli elementi.

2.5.1 La chiave

La chiave è la prima mossa del bianco ed è quella porterà alla svolta decisiva del gioco: il matto al nero. Salvo che il compositore non specifichi il contrario, deve essere unica. Una chiave può introdurre una o più minacce di matto.
La chiave è considerata tanto più pregevole quanto più riesce nascosta ed imprevedibile, deve essere quindi il meno aggressivo possibile. Ne conseguono le regole per una buona chiave: non deve dare scacco, non deve catturare figure avversarie (fanno eccezione i pedoni), non deve togliere case di fuga al Re (almeno che non ne conceda altre), non deve mettere in gioco pezzi inattivi del proprio schieramento; tutte queste regole hanno le loro eccezioni.

2.5.2 La minaccia

E' la mossa del bianco che darebbe matto se eseguita dopo la chiave purché al nero non venga permesso di muovere. Ce ne possono essere diverse. Molti problemi si basano proprio sulla presenza di più minacce (tema Fleck). Esiste inoltre una categoria di problemi complementari in quanto non hanno alcuna minaccia. Tali problemi vengono definiti "a blocco".

2.5.3 Le difese

Sono le possibili mosse che il nero ha a disposizione per cercare di sventare il matto. Più sono numerose, più il problema è interessante. La difesa è pesantemente vincolata dalla chiave del problema.

2.5.4 *Il matto*

Il matto è la mossa conclusiva del bianco ossia quella che permette la sconfitta del nero. Non è detto che debba esistere un solo modo per dare scacco matto, ma deve essere garantito che ad ogni possibile difesa nera esista una mossa bianca mattante. Se esistono più mosse mattanti per una stessa difesa, allora il matto viene definito **duale**. Un matto di questo genere degrada la qualità della composizione.

Un problema che ha soluzioni diverse da quelle previste dall'autore, si dice **demolito** e non ha più alcun valore. Fra i problemi demoliti ricadono anche quelli che hanno soluzioni in un numero inferiore di mosse rispetto all'enunciato.
Un problema che non ha alcuna soluzione si dice **insolubile**.

2.5.5 *Esempi*

Per chiarire meglio il significato di questi elementi, prendiamo in esempio il semplicissimo problema in figura 3:

2.5.5.1 Esempio 1 - Matto in 2

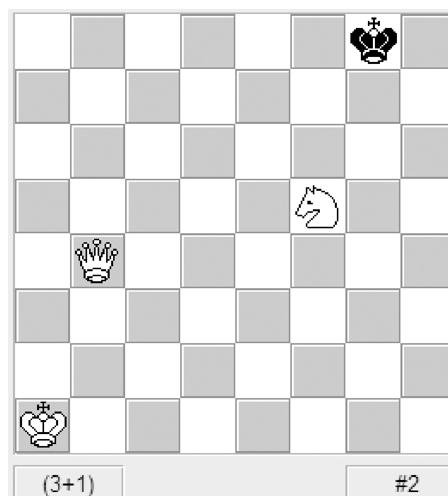


Figura 3: Regina e cavallo - matto in 2

In questo problema sono presenti 3 pezzi bianchi ed un solo pezzo nero (da qui la scrittura 3+1 in fondo all'immagine).
La chiave di questo problema è De7 che introduce la minaccia Dg7#.

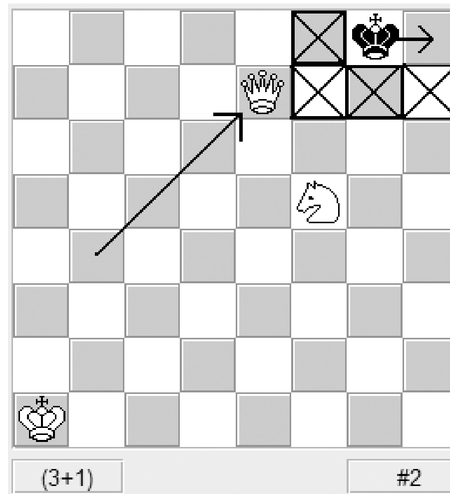


Figura 4: la chiave e le vie di fuga del re

Grazie alla chiave, le possibili difese del Re nero si riducono a Rh8.
Il matto del bianco giunge con la mossa Dg7#.

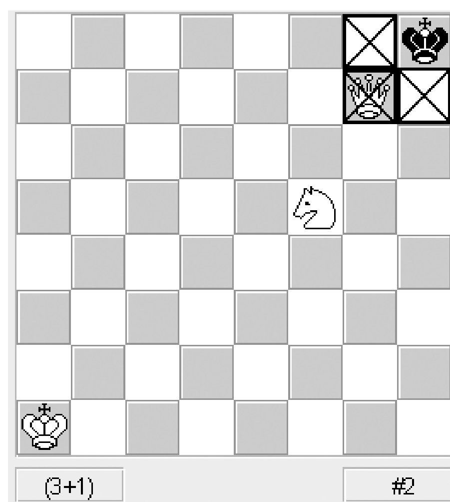


Figura 5: la regina da scacco matto al re nero

Quindi:
1. De7! {2. Dg7#}
 1. ... Rh8 2. Dg7#

La coppia 1…Rh8 2. Dg7# viene chiamata **variante** ed è costituita dalla difesa nera e dal corrispondente matto. In questo esempio ne esiste una sola, ma possono essere molte di più a beneficio della qualità del problema.
Osservando la notazione, emerge che la minaccia coincide con il matto. Ciononostante, non è affatto vero che tutti i matti di un problema debbano corrispondere alle minacce. Anzi, nella maggior parte delle composizioni, non accadde quasi mai. Si pensi inoltre che esiste anche una categoria di problemi che non hanno alcuna minaccia. Ciò non significa che tali problemi debbano essere insolubili.

2.5.5.2 Esempio 2 - Blocco

Come anticipato nella descrizione degli elementi di un problema, esistono composizioni che non possiedono alcuna minaccia e che quindi necessitano di una mossa del nero per poter dare matto con la seconda mossa. Il bianco costringe in "zugzwang" l'avversario. Questi problemi vengono quindi definiti "a blocco".
A titolo d'esempio, si consideri il problema di Josef Cumpe (Bohemia 1908) qui sotto raffigurato.

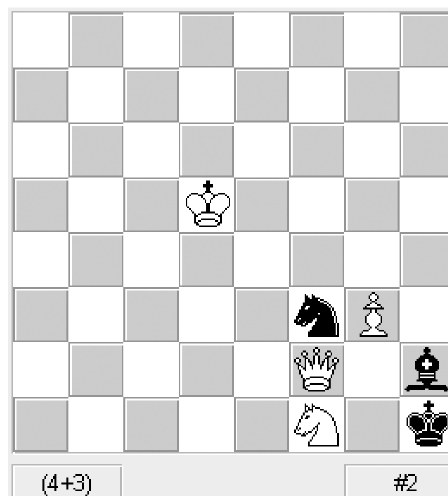


Figura 6: Josef Cumpe (Bohemia 1908)

Qualunque giocatore probabilmente muoverebbe 1.D:f3, ma ai fini del problema non ha alcuna importanza. Dopo 1.D:f3 il nero potrebbe rispondere con Rg1 e il Bianco non potrebbe più mattare in una sola mossa. D:f3 quindi non è la chiave.
Se toccasse al Nero muovere, potrebbe giocare:

- Ag1, ma il Bianco matterebbe con Df3#;
- Ag3,ma il Bianco matterebbe con C:g3#;
- Cf3-ovunque, ma il Bianco matterebbe con D:h2#;

Dunque se il Bianco potesse fare una mossa "neutra", potrebbe mattare in due mosse.

Il pezzo più estraneo al gioco, è il Re. Spostandolo in d6, viene ripristinato lo stato descritto precedentemente e sarà possibile ottenere lo scacco matto in 2 mosse.

Riassumendo:
La chiave è Rd6. E' unica, poiché spostando Rc6, Rc4, Re6, Re4 lo si esporrebbe alla minaccia del cavallo Nero. Rc5 è una mossa legale, ma con Ag1 la regina verrebbe inchiodata e non potrebbe dare matto.

Il problema ha le seguenti varianti:

Difesa	**Matto**
Ag1	D:f3#
A:g3	C:g3#
Cf3 ovunque (g1,e1,d2,d4,e5,g5,h4)	D:h2#

Vengono ora presentati due problemi non validi.

2.5.5.3 Esempio 3 - Nessuna soluzione

Come detto precedentemente, il bianco deve dare scacco matto come conseguenza di ogni difesa nera.
Osservando il problema di figura 7, vediamo che muovendo il Re in g6, introduciamo la minaccia di matto Te8#.

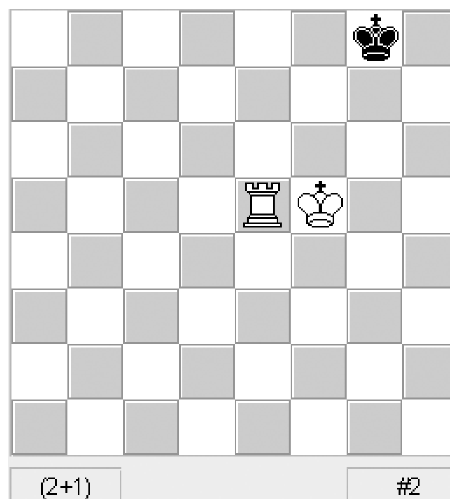


Figura 7: Re e torre, nessuna soluzione

Quindi utilizzando Rg6 come chiave, il nero beneficerebbe di solo due difese, ossia:

Rh8

Rf8

Tuttavia una sola di queste lo porterà al matto del bianco. Se il Re nero si spostasse in h8, il bianco matterebbe con Te8#.

In caso contrario, la mossa Rf8, si dimostrerebbe tale da sventare la minaccia nemica, poiché alla torre bianca verrebbe impedito di mattare.

L'esistenza di una difesa nera in grado di evitare il matto, rende la chiave inefficace e poiché non ne esistono altre, il problema non ammette soluzioni, quindi è insolubile.

2.5.5.4 Esempio 4 - Più soluzioni

Un altro problema non valido è quello costituito da più chiavi che portano al matto. Un esempio evidente ne è il problema in figura 8.

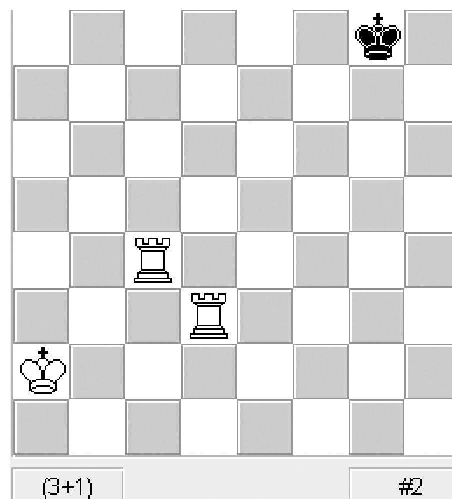


Figura 8: Due torri e più soluzioni

Sulla scacchiera sono presenti 2 torri bianche che possono raggiungere la settima traversa senza che il loro tragitto venga ostacolato da altri pezzi. Nessun pezzo nemico, inoltre, è in grado di minacciare le loro case di destinazione. Una torre posizionata in settima traversa comprometterebbe seriamente la situazione del Re nero lasciandogli solo 2 vie di fuga, f8 e h8. Indipendentemente dalla scelta, la seconda torre matterebbe raggiungendo l'ottava traversa.

Poiché è indifferente quale delle due torri raggiunge la settima traversa e quindi quale matta, il problema ha 2 soluzioni e viene perciò considerato demolito.

Soluzione: 1. Tc7 [2. Td8#]
1. ... Rf8/h8 2. Td8#

Soluzione: 1. Td7 [2. Tc8#]
1. ... Rf8/h8 2. Tc8#

2.5.5.5 Esempio riassuntivo

Prendiamo ora in considerazione il seguente problema:

In questa composizione, vengono presentate tutte le caratteristiche che un problema dovrebbe o non dovrebbe avere. Come vedremo è molto semplice anche perché include un esiguo numero di pezzi. Alla radice di questo albero, troviamo il problema così come il compositore l'ha creato. Sotto la prima scacchiera troviamo le possibili prime mosse del bianco. Per motivi di spazio non sono state menzionate tutte, ma solo le più significative.
Candidate chiavi come Ra2, Te4 o Th2 non sono state approfondite perché non contengono elementi di particolare interesse. Tuttavia sono stati indicati 3 rami che simboleggiano le 3 uniche difese nere. Proseguendo per ognuno di quei rami, non otterremmo nessun matto perciò, ho risparmiato spazio per varianti di maggior valore.

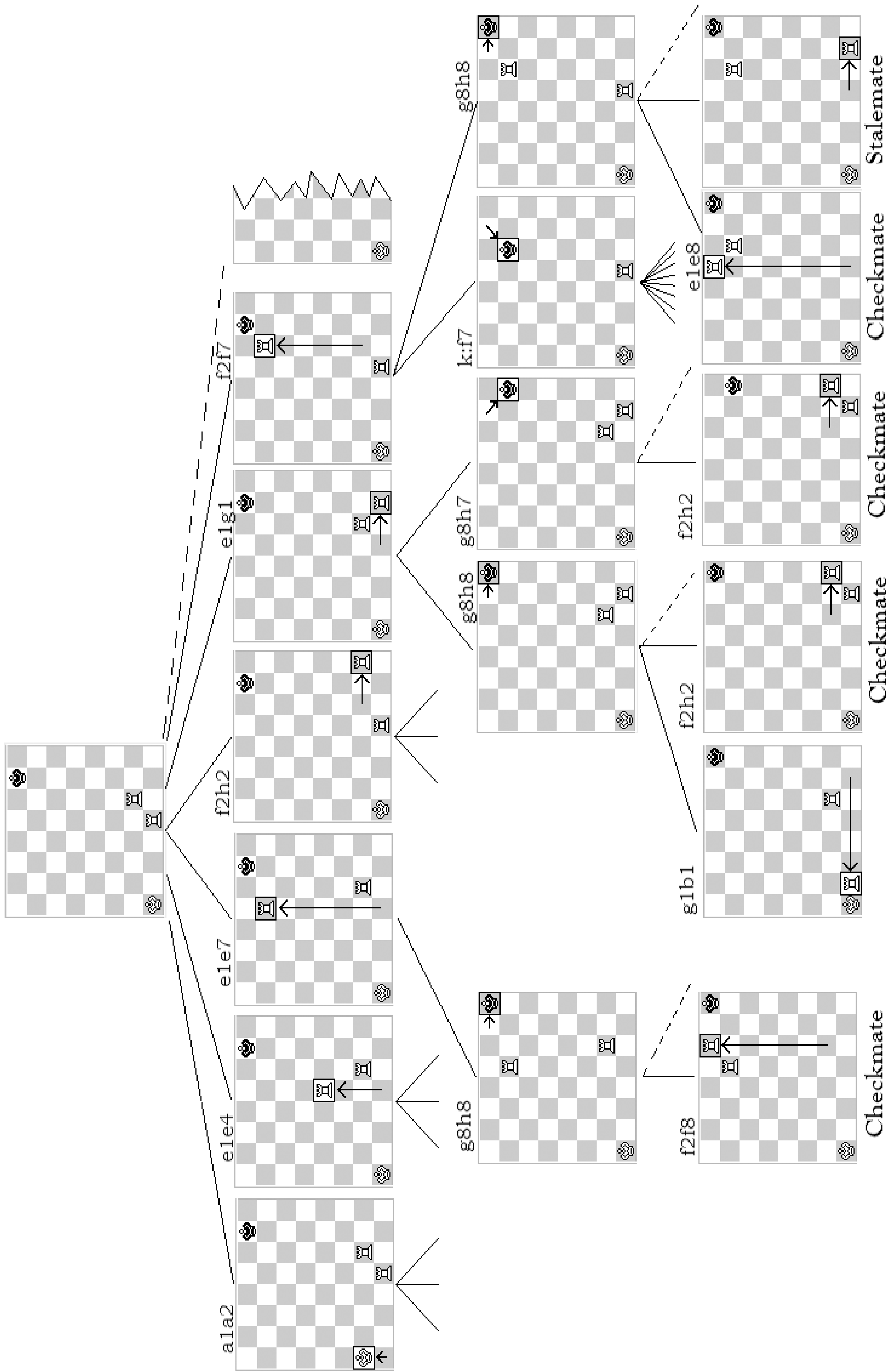
Partiamo dalla prima mossa bianca Te7. Le possibili mosse del nero passano da 3 ad 1. Il Re nero si vede quindi costretto ad eseguire l'unica difesa (Rh8). Ora è molto semplice per il bianco dare scacco matto al nero, poiché è sufficiente spostare la torre da f2 a f8. Ovviamente il bianco ha a disposizione diverse possibili mosse, ma solo Tf8 è in grado d mattare. Possiamo quindi affermare che la mossa iniziale Te7 è una chiave!

La prima mossa del bianco Tg1 vieta al Re nero di accedere alla casa g7 permettendogli di fuggire solo in due modi, ossia Rh8 ed Rh7. Ad ognuna di queste difese, il bianco sa rispondere con un matto muovendo Th2. Il fatto che esista uno stesso matto per due difese differenti, è un'imperfezione del problema. Ad ogni modo possiamo ritenere che la mossa Tg1 sia anch'essa una chiave.

Il caso più interessante è quello in cui il bianco compia come prima mossa Tf7. In questa situazione, il Re può raggiungere 2 case: h8 ed f7. Nel primo caso, Te8 concluderebbe la partita, mentre un errore del bianco come Tg1, farebbe guadagnare uno stallo al Re nero.
Nel secondo caso invece, il Re nero può catturare la torre in f7 non permettendo lo scacco matto in due mosse. Ora, abbiamo visto che una delle due difese del nero, si dimostra inefficace, mentre l'altra eviterebbe il matto. In conclusione non possiamo accettare Tf7 come chiave.

Il problema appena visto, oltre che essere molto semplice, è anche di pessima qualità, poiché:

1. esistono più chiavi (problema demolito)
2. esiste uno stesso matto per due difese differenti
3. esistono poche varianti


f2f7
e1g1
f2h2
e1e7
e1e4
a1a2
g8h8
k:f7
g8h7
g8h8
g8h8
e1e8
f2h2
f2h2
g1b1
f2f8
Stalemate
Checkmate
Checkmate
Checkmate
Checkmate

Problemi e qualità

All'interno della FIDE si è creata una specifica commissione che si occupa di gestire competizioni scacchistiche a livello internazionale. Questo organo si chiama PCCC (Permanent Commission of the Fide for Chess Composition) e attualmente dipende da federazioni scacchistiche situate in 39 stati. La PCCC si occupa di organizzare tornei internazionali per compositori e per risolutori, di definire linee guida per ogni genere di problema, di divulgare informazioni di pubblico interesse e di diffondere in tutto il mondo la passione verso la problemistica. Ogni due anni inoltre propone i campionati mondiali per tutti gli interessati.[4]
All'interno di ogni competizione, viene fatta una valutazione del problema per stabilirne le qualità. I parametri secondo i quali un problema è migliore di un altro sono diversi, ma tutti puntano alla "spettacolarità" della soluzione.
I problemi devono essere in grado di affascinare il risolutore e sono tanto più graditi quanto ricchi di combinazioni. Maggiori sono le varianti più il problema diventa interessante, ma anche difficile da progettare.
Come detto precedentemente, la chiave deve essere il più nascosta possibile ed imprevedibile. Inoltre la possibilità di più matti per una stessa difesa inficia pesantemente la qualità del problema.
Un problema dovrebbe avere qualità artistiche, estremamente difficili da definire oggettivamente e mutevoli negli anni. In ogni caso ogni problema dovrebbe svolgere un tema e non essere solamente difficile da risolvere. Fino al XIX secolo la difficoltà era considerata un pregio del problema, così che molte posizioni erano affollate di pezzi inutili, che servivano unicamente a rendere più difficile la soluzione. Oggi un pezzo inutile (che cioè può essere tolto dalla posizione senza cambiarne la soluzione) è considerato un difetto molto grave in un problema.[1]

il programma

1. INTRODUZIONE

Comprendere a fondo l'attività del compositore è stato tutt'altro che banale, sebbene conoscessi le regole del gioco degli scacchi da molto tempo. D'altro canto, non avrei mai potuto iniziare e proseguire il lavoro della tesi senza tener conto costantemente dello scopo finale e di tutti quegli elementi di cui l'utente ultimo ha bisogno.
Ora che è stata data un'infarinatura di quella che dovrebbe essere l'attività del compositore, vediamo come il mio applicativo risponde alle esigenze del destinatario.

2. IL PROGRAMMA

Il programma che risolve problemi di scacco matto #2 è stato scritto interamente in linguaggio C. Tale scelta è stata vincolata dall'esistenza di un precedente software che si occupa di gestire competizioni fra umani e/o motori. Completare un programma partendo dal nulla avrebbe richiesto tantissimo tempo sebbene molte parti necessarie per lo sviluppo non siano di particolare difficoltà implementativa. Per risparmiare tempo, si è pensato di riutilizzare codice esistente per le funzioni comuni fra il gioco attivo e calcolo dello scacco matto in due mosse. Benché le due modalità di gioco siano molto differenti, esistono numerosi elementi generali. Si pensi ad esempio alla generazione delle mosse legali dei pezzi oppure l'avvicendamento dei turni che, malgrado si riduca a due soli cambi, segue l'usuale logica.
Esistono poi una serie di elementi di contorno che possono essere modificati e riutilizzati per rendere il programma più ricco di informazioni e più usabile. Fra questi troviamo la funzione di stampa che mostra a video la scacchiera, il ciclo utilizzato per la gestione degli input dell'utente, la variabile utilizzata per il conteggio dei nodi visitati dell'albero delle mosse.
Il lavoro svolto è stato quindi quello di modificare un programma esistente e di adattarlo alle proprie esigenze. Si è perciò dovuto manipolare gli input per adeguarli alle strutture dati predefinite, modificare profondamente l'algoritmo per la scansione dell'albero delle mosse ed infine generare l'output desiderato secondo un formato opportuno per compositori di problemi scacchistici.
Tutto questo ha richiesto una discreta conoscenza delle nozioni scacchistiche essenziali per l'attività del compositore e la comprensione approfondita dei paradigmi della “chess programming”.

2.1 SCHEMA

L'immagine seguente indica le varie parti di cui è composto il programma e le relazioni che ha con il mondo esterno.

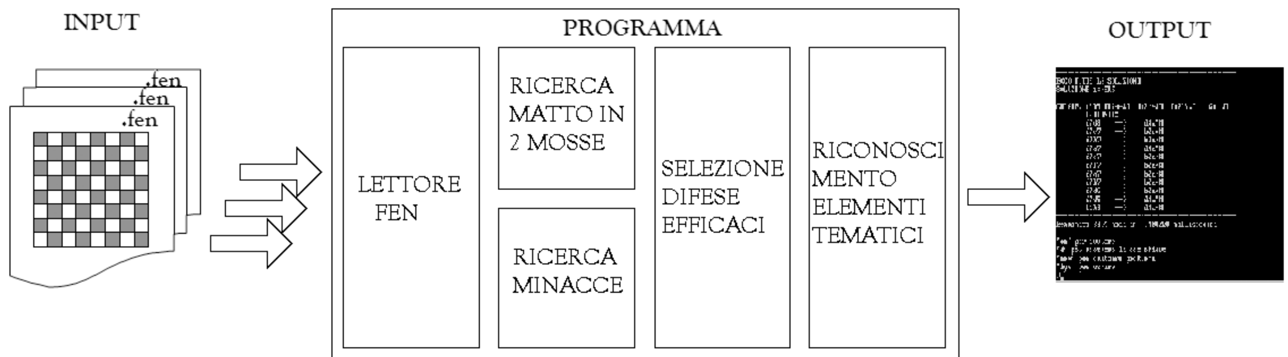


Figura 1: Schema a blocchi del programma

Si noti in particolare la modularità: abbiamo 3 blocchi fondamentali di cui il primo e l'ultimo sono rispettivamente l'input e l'output del programma.
Dal punto di vista dell'utente, questo schema segue il modello black-box in quanto si può immaginare di essere a conoscenza del comportamento della "scatola nera" senza sapere cosa vi è all'interno. D'altro canto, per lo sviluppatore il programma è visto come un insieme di moduli cooperanti per raggiungere uno scopo. Il fatto che il programma sia suddiviso in sottoprogrammi (talvolta un sotto programma è formato da una sola funzione), rende più semplice la comprensione nonché lo sviluppo stesso del software.

2.1.1 Descrizione

Il primo blocco all'inizio della catena è costituito dal file che il compositore desidera prendere in esame. L'estensione specificata fra parentesi tonde, indica che il file deve seguire un particolare formato affinché possa essere interpretato correttamente. Il file in questione è residente su disco e deve essere situato nella cartella dove giace il programma per non dover specificare l'intero path.
Il primo blocco del programma si occupa di aprire il file di input ossia tutte quelle informazioni riguardanti il problema da analizzare. Questa parte di codice è stata creata personalmente senza il supporto di nessun codice preesistente.
Dopo aver interpretato correttamente le informazioni contenute nel file di input, il primo blocco deve generare degli output indispensabili ai blocchi successivi, ovvero deve predisporre i dati nelle variabili su cui gli algoritmi svolgeranno il loro compito.
I due blocchi successivi sono essenzialmente due funzioni che percorrono l'albero delle mosse e salvano i risultati in nuove strutture dati che verranno utilizzate in tutte le lavorazioni successive. E' proprio in questo punto che il programma processa l'input e trova le soluzioni del problema, nonché buona parte delle informazioni spiegate nel corso del capitolo.
Il quarto blocco esegue operazioni di raffinamento degli output ottenuti nei precedenti due bloc-

chi, in quanto rimuove soluzioni che non farebbero altro che creare confusione all'utilizzatore. Benché possa sembrare piuttosto semplice, questo blocco richiede la creazione di un nuovo albero di mosse legali, ma i dettagli verranno spiegati in seguito.
L'ultimo blocco del programma lavora sui dati generati dall'algoritmo di ricerca del matto in 2 mosse e svolge banali operazioni di confronto per poter riconoscere particolari pattern che potrebbero agevolare il lavoro del compositore.
L'ultimo blocco dello schema non è altro che l'insieme degli output provenienti dalle elaborazioni svolte dal programma. Nessun file verrà scritto in questa fase, ma le informazioni verranno semplicemente visualizzate attraverso un terminale.

Nei prossimi paragrafi verranno presentati i singoli blocchi nel dettaglio.

2.2 Il file

Il file da passare in input al programma per il calcolo dello scacco matto #2 deve contenere tutte le informazioni relative ai pezzi in modo da poter formare lo **stato iniziale** del gioco. Non è tutto qui: oltre alla posizione di ogni pezzo appartenente al problema, è necessario notificare la possibilità di arrocchi e di prese en-passant associate ad entrambi gli schieramenti.
Per codificare tutte queste informazioni, è sufficiente del testo opportunamente formalizzato e che, se possibile, segua precisi standard.

2.2.1 Fen

Fra i possibili standard impiegati in ambiti scacchistici, ho pensato di utilizzare il formato FEN (Forsyth-Edwards Notation). E' molto semplice anche se racchiude un esiguo contenuto informativo. Esso tratta solo ed esclusivamente lo stato della partita eliminando il vincolo sulla storia, poiché non tiene traccia delle mosse precendenti che hanno portato alla situazione descritta nella notazione. Questa potrebbe essere una limitazione se si ha a che fare con gioco attivo e per questo motivo la notazione FEN è molto meno usata, ma nel mio caso calza a pennello. E' infatti possibile ottenere un file valido senza preoccuparsi di come si è arrivati a quelle posizioni, permettendo un editing più rapido al compositore e risparmiando indubbiamente sulle operazioni che la macchina deve compiere per caricare il problema.

Il formato FEN è caratterizzato da sei campi separati da uno spazio. Li riassumo brevemente.
1)Le posizioni dei pezzi:
Viene presentato il contenuto di ogni casa ordinato per riga partendo da a8-h8 fino ad a1-h1. Ogni riga è separata dalla successiva grazie ad un carattere speciale (/). Il contenuto di una casa è la lettera iniziale del pezzo che vi risiede. Se il pezzo è bianco si utilizzano le lettere maiuscole.
Quindi:

	NERO	BIANCO
PAWN	p	P
BISHOP	b	B
KNIGHT	n	N
ROOK	r	R
QUEEN	q	Q
KING	k	K

A differenza di quanto detto per le case possedute da pezzi, le case vuote vengono indicate con delle cifre che indicano quante ve ne sono in successione.
Per chiarire quanto visto finora, viene mostrato come verrebbe rappresentato il primo campo della notazione dopo la mossa “e2e4” di una comune partita:

rnbqkbnr/pppppppp/8/8/4P3/8/PPPP1PPP/RNBQKBNR

Se volessimo effettuare primo rapido controllo della validità di questo campo, dovremmo fare la somma dei valori contenuti in ogni sottocampo e verificare che il totale sia esattamente pari ad 8. Ovviamente il contributo delle lettere è di uno. Un secondo controllo dovrebbe garantire che le lettere impiegate siano le 12 ammesse (6 minuscole e 6 maiuscole) e che le cifre ammesse siano comprese tra 1 e 8.

2)Il secondo campo dice semplicemente chi deve muovere, se il bianco o il nero. Una lettera minuscola è sufficiente ossia “w” per il bianco e “b” per il nero.
Dopo “e2e4” del bianco avremo:

rnbqkbnr/pppppppp/8/8/4P3/8/PPPP1PPP/RNBQKBNR b

visto che il bianco ha mosso ed ora è il turno del nero.
Poiché l'impiego è nell'ambito dei problemi di matto #2, bisogna verificare che questo campo sia effettivamente pari a “w” altrimenti andrebbe ignorato. E' risaputo che in questo genere di problemi, il bianco è il primo ad effettuare la mossa, quindi spetta a lui muovere. Posso concludere che questo campo non sia rilevante ai miei fini mentre è fondamentale se si deve riprendere una competizione di gioco attivo.

3)Il terzo campo evidenzia le possibilità di arrocco dei contendenti. Prima vengono presentati gli arrocchi bianchi poi quelli neri. Qualora non vi siano possibilità di arrocco, il semicampo viene marcato con “-“. L'arrocco lungo viene indicato con q o Q per specificare che è dal lato della regina, mentre quello corto con k o K per specificare che è dal lato del re. Nel caso d'esempio, poiché la partita è solo alla prima mossa, avremo questa sequenza:

rnbqkbnr/pppppppp/8/8/4P3/8/PPPP1PPP/RNBQKBNR b KQkq

4)Il quarto campo indica quale è la casa su cui il nemico può catturare con una presa en-passant. Il campo è costituito da una coppia “colonna-riga” e nel nostro caso d'esempio viene indicato così:

rnbqkbnr/pppppppp/8/8/4P3/8/PPPP1PPP/RNBQKBNR b KQkq e3

Dato che le prese en-passant possono avvenire a seguito dell'avanzata di due case di un qualsiasi pedone, ci si attende che la cifra di questo campo sia o “3” o “6”. Le prese en-passant esistono solo per la terza e per la sesta traversa, indipendentemente dalla colonna. Queste informazioni sono utili per determinare la correttezza di un file in notazione FEN.

5)Il quinto campo è un valore intero non negativo ed è il contatore delle semimosse (*ply*). Una semimossa è il movimento di un pezzo da parte di uno dei due giocatori. Due semimosse, una bianca e una nera, costituiscono la mossa nel senso tradizionale. Questo campo è utile nelle competizioni in cui vige la regole delle 50 mosse.

6)L'ultimo campo è un contatore di mosse che viene aggiornato subito dopo la semimossa del nero, quando la mossa si completa. Non trovo particolare utilità se non per indicare quanto è stata lunga la partita fin a questo punto. Evidentemente un conteggio delle mosse o delle semimosse non sono utili ai fini e possono essere ignorati.

Con tutti questi elementi sono riusciti a ricostruire lo stato di una partita pur non avendo a disposizione la sequenza degli avvenimenti precedenti. Vedere le posizioni dei pezzi in un problema risulta pressoché immediato.

2.3 Lettore fen

Un programma per il gioco degli scacchi permettere ad un utente di sfidare un avversario, umano o artificiale che sia, in una competizione scacchistica tramite l'utilizzo di comandi da tastiera o con il supporto di un'interfaccia grafica. Oltre verificare l'ammissibilità delle mosse che l'utente propone, tale software deve anche gestire tutte le situazioni che si possono presentare durante una qualunque partita di scacchi, come lo scacco matto, la gestione degli orologi e, nel caso in cui uno dei due giocatori sia artificiale, deve sapere anche scegliere quale mossa gli porterebbe maggior vantaggio.
Tuttavia, a meno che non si tratti di un'applicazione specifica, un programma per il gioco degli scacchi non è in grado di avviare competizioni con modalità di gioco differenti (*varianti*). Esso è predisposto per iniziare un'elaborazione partendo da una precisa configurazione di pezzi sulla scacchiera, la solita del gioco attivo, ossia neri e bianchi separati rispettivamente nelle prime due ed ultime due traverse della scacchiera.
E' evidente che questa configurazione di pezzi non ha più alcun significato quando si passa nel mondo della problemistica, per diverse ragioni. La più ovvia è che ogni problema di scacco matto

#2 ha una ubicazione dei pezzi differente e non una di partenza comune ad ogni problema. La seconda è banale: la tipica configurazione del gioco attivo non ammette soluzioni, quindi non è di particolare interesse per i compositori.
Nasce così l'esigenza di integrare al programma una porzione di codice in grado di leggere da file la configurazione iniziale di pezzi che si intende analizzare. Il programma, non è dotato di interfaccia grafica che permetta la visualizzazione di finestre di dialogo per l'apertura di file. E' possibile tuttavia esprimere quale problema si voglia analizzare al lancio dell'eseguibile oppure tramite comando a programma già avviato. L'unico vincolo è che il file risieda nella stessa directory del programma, altrimenti va specificato anche l'intero path. Il nome del file non deve necessariamente rispettare i caratteri maiuscoli e minuscoli, tuttavia non va dimenticata l'estensione (.fen)

Figura 2: Apertura di un file

Qualora il compositore decida di passare ad un altro problema, non è necessario riavviare il programma. Fra le opzioni che l'utente può selezionare, esiste anche il comando “new” che permette di aprire un nuovo file. Una volta digitato il comando, verrà fatta richiesta all'utente di inserire il nome del problema, proprio come segue:

Figura 3: Apertura di un nuovo file

2.3.1 Il codice

Nel paragrafo 2.2 abbiamo definito l'input del programma, ma ora serve una porzione di codice in grado di interpretare le informazioni e di aggiornare le strutture dati su cui verranno effettuate le elaborazioni degli algoritmi.
Il codice che legge file in formato FEN deve quindi saper aprire il file richiesto, suddividere il testo in campi ed analizzare il contenuto di ciascuno. Una volta prelevate le singole informazioni, deve poi indirizzarle alle variabili che il programma per il gioco degli scacchi già possiede per le consuete computazioni. Possiamo immaginare la funzione come suddivisa in 2 blocchi principali.
Il primo si occupa sostanzialmente dell'accesso in lettura al file e di memorizzare i dati letti in un vettore, in modo da rendere le elaborazioni più rapide.
Il secondo blocco analizza l'intero vettore ed in base al contenuto dei singoli elementi organizza le variabili che verranno utilizzate dagli algoritmi. Queste operazioni piuttosto semplici, richiedono circa 150 istruzioni divise in 4 sottoblocchi che operano sui soli campi utili. Il codice include anche dei controlli basilari che determinano se il problema può essere valido. Non è possibile infatti creare un problema in cui compaiono più re di uno stesso colore oppure un problema in cui non ve ne sono. Le caratteristiche che un gioco #2 dovrebbe avere sono già state specificate nel capitolo precedente.
Riporto ora la funzione che apre il file e ne legge le informazioni necessarie. Ovviamente non sono presenti tutte le istruzioni utilizzate, ma solo quelle fondamentali.

```
BOOL ApriFile(Scacchiera * scac, char NomeFile[]){

     char collocazioni[RIGA];
     char parziale[RIGA];
     int counter;
     int indice;
     int campo;FILE *FileInput;
     char carattere;
     char duelettere[2];
     BOOL esisteRenero=FALSE,esisteRebianco=FALSE;


       printf("APRO IL FILE: %s\n",NomeFile);
```

Dopo aver dichiarato le variabili necessarie per l'apertura del file, il programma segnala all'utente di aver capito quale file si intende analizzare.

```
     FileInput = fopen(NomeFile,"rt");
```

Ora è possibile procedere con l'apertura del file in modalità “read” poiché non è necessario doverlo modificare. FileInput è un puntatore a file, ossia una variabile il cui contenuto è l'indirizzo della struttura FILE. Specificando la modalità di apertura "rt", si indica che il contenuto del file è di solo testo.
La `fopen()` è una funzione che ritorna un valore NULL se non va a buon fine per una qualsiasi

ragione: memoria esaurita, file inesistente ecc. Per questo motivo viene effettuato questo controllo:

```
 if (FileInput == NULL){
          printf("File %s FILE INESISTENTE!\n",FileInput);
           printf("DIGITARE NUOVO NOME\n");
          return FALSE;
     }
```

Ovviamente, se l'apertura fallisce deve essere segnalata all'utente. La funzione ritorna un valore di errore, ma rilancia la possibilità di riscrivere il nome del file correttamente o di specificarne un altro.

```
     indice=0;
      while(
fscanf(FileInput,"%c",&collocazioni[indice])!=EOF){
           indice++;
      }
      fclose(FileInput);
```

Qualora l'operazione di apertura del file vada a buon termine, viene immediatamente fatta la lettura del contenuto. La funzione `fscanf()` permette di leggere dal file un carattere alla volta. Il valore acquisito viene copiato in un vettore.
Il ciclo viene interrotto quanto si raggiunge la terminazione del file.
Ora il file di input non è più necessario poichè i dati utili sono già stati copiati in memoria. La `fclose()` provvede alla chiusura del file.

La seconda parte di questa funzione ha il compito di suddividere le informazioni copiate nel vettore e quindi di trasferire il contenuto informativo in nuove strutture dati adatte alle necessità dell'algortimo di calcolo.
Sostanzialmente consiste in quattro porzioni di codice, una per ogni campo utile della notazione FEN.
Il primo blocco è formato da un ciclo che scandisce il vettore finchè non incontra il carattere separatore della notazione: lo spazio.
I blocchi successivi hanno lunghezza prevedibile, o comunque limitata a pochi caratteri e quindi non ho utilizzato alcun ciclo. Una volta suddivisi i campi e prelevate le informzioni di ciascuno, viene effettuata la chiamata della funzione AggiornaScacchiera(campo,parziale,scac) che riceve in ingresso, il campo di riferimento, le informazioni ad esso associate e la struttura dati da aggiornare.
All'interno di ogni blocco sono presenti alcuni controlli che verificano l'ammissibilità del campo. In particolare si controlla che esista un solo re per ogni schieramento.

```
while(collocazioni[counter]!=' '){
           carattere=collocazioni[counter];
           if(carattere!='/'){
                 parziale[indice]=collocazioni[counter];
          if(carattere == 'k') {
               if(!esisteRenero)
                    esisteRenero=TRUE;
               else{
                    printf("ERRORE: PIU' RE SULLA SCACCHIE-
RA");
                    exit(EXIT_OPENFILE);
                    }
                              }

...
 }else{

                 AggiornaScacchiera(campo,parziale,scac);
                    SvuotaArray(parziale,RIGA);
                 campo++;
                 indice=-1;
           }
           counter++;
           indice++;
     };

     AggiornaScacchiera(campo,parziale,scac);
      counter++;
```

2.4 La scelta del motore

Fin qui si è trattato solo del perché è necessario disporre di un programma di scacchi e quindi del suo ruolo nell'ambito del calcolo dello scacco matto #2. Tuttavia non è ancora stato illustrato quale serve e quali debbano essere le sue caratteristiche. Requisito fondamentale è che sia disponibile il codice sorgente per apportare le consistenti modifiche richieste e per raggiungere lo scopo finale. La scelta è ricaduta su TSCP [5], un software scaricabile gratuitamente da internet. I dettagli di questo applicativo verranno superficialmente descritti nel paragrafo successivo.

2.5 TSCP (Tom Kerrigan's Simple Chess Program)

2.5.1 Breve descrizione

Come dice il nome stesso, TSCP è un semplice programma per il gioco degli scacchi. Esso è stato concepito per chi si imbatte per la prima volta nella chess programming. E' scritto in linguaggio C e non conta più di 4000 righe. Leggendo il codice ci si accorge di quanto sia facile comprendere quello che il programma fa, in ogni punto o situazione del gioco.
TSCP permette competizioni scacchistiche fra due utenti o fra utente e macchina. Non gode di interfaccia grafica propria, ma è Winboard compatibile ossia è in grado di comunicare con gli elementi grafici del celeberrimo software scritto da Tim Mann e supportato da numerosissimi altri motori. L'utente TSCP può scegliere fra numerosi livelli di difficoltà (fino a 32) se deve affrontare un giocatore artificiale.
Come già anticipato, TSCP è un software straordinariamente semplice da comprendere. Utilizza strutture dati relativamente banali e il codice include tutto e solo il necessario per poter sostenere competizioni scacchistiche.
Il software è coperto da Copyright e non può essere distribuito.

2.5.2 Dettagli TSCP

In questo paragrafo, verranno presentati alcuni dettagli sul funzionamento di TSCP. Non è mia intenzione spiegare il programma in ogni punto, ma un breve riepilogo potrebbe aiutare a capire le modifiche fatte e quindi le vie scelte per raggiungere l'obiettivo finale.
In TSCP la generazione delle mosse ammissibili, avviene attraverso la funzione `gen()` che dopo aver osservato a chi tocca giocare, accede ad un insieme di matrici le quali permettono di stabilire quali sono le case raggiungibili da ogni pezzo di uno schieramento. Una serie di controlli determinano se una casa di destinazione o appartenente al tragitto del pezzo, è libera oppure occupata da un pezzo nemico o amico.
Le strutture dati utilizzate sono principalmente matrici predefinite, fra cui:

mailbox[120]
mailbox64[64]

che si occupano esclusivamente di determinare dove un pezzo può andare.

slide, offsets, e offset

sono le direzioni in cui i pezzi possono muoversi.

castle_mask[64]

gestisce gli arrocchi.

init_piece[64]

che contiene le posizioni iniziali di una partita.

Le mosse generate vengono salvate attraverso una funzione `gen_push()` in un vettore(gen_dat[]), mentre le sequenze di mosse, vengono memorizzate in una struttura a pila(hist_dat[]), in modo da poter ricostruire la successione degli avvenimenti in caso di necessità. Tutte queste strutture dati sono sufficienti per poter appurare la correttezza di una partita a scacchi.
Ora però occorre trattare la parte più delicata ma allo stesso tempo attraente del programma: la gestione del giocatore artificiale.

Il motore TSCP infatti deve saper decidere quale mossa fra le legali è meglio intraprendere in risposta a agli attacchi nemici tenendo conto dell'accuratezza richiesta dall'utente. L'utente può decidere quanto debba essere abile il suo nemico artificiale in modo da poter meglio soddisfare il proprio piacere per il gioco. Quindi, TSCP fornisce la possibilità di modificare le proprie attitudini. Tale feature si traduce in una valutazione di una mossa ad un profondità superiore od inferiore nell'albero di esplorazione delle mosse legali. Ci si attende che un giocatore artificiale con superiori abilità richieda maggior tempo per decidere quale mossa effettuare poiché dovrà raggiungere nodi di profondità più elevata. Giunti alle foglie, una funzione `eval()` si occuperà della vera e propria valutazione della posizione seguendo tabelle contenenti i valori arrecati da ogni pezzo. La scelta della mossa migliore è determinata da diversi parametri, tra cui il valore associato ai pezzi, la loro posizione sulla scacchiera e rispetto agli altri pezzi.
La bontà di una mossa non può essere stabilita giudicando solo la posizione immediatamente ottenuta, ma è necessario prevedere la posizione finale di una serie di mosse costituita dalle migliori risposte dell'avversario e possibili contromosse del programma.
L'insieme delle varianti costituisce un albero e la scelta della mossa migliore richiede l'identificazione del percorso ottimo.
Al fine di non testare ogni possibile mossa(impossibile in tempi ragionevoli per gli attuali calcolatori se i livelli sono tanti), il motore di TSCP implementa l'algoritmo Alpha-Beta per l'eliminazione delle mosse poco vantaggiose. L'intera logica seguita per la scelta della mossa migliore è contenuta nella funzione `search()`.
La funzione di ricerca `search()` non lavora su alcuna struttura dati, bensì crea l'albero delle mosse implicitamente attraverso una serie di chiamate ricorsive.
TSCP infine mette a disposizione funzioni per il calcolo delle prestazioni sulla macchina su cui viene avviato.

2.5.3 TSCP e #2

TSCP, come la maggior parte dei programmi per il gioco degli scacchi, possiede numerose funzionalità di cui molte non necessarie quando si ha a che fare con il calcolo dello scacco matto #2.
Come già trattato nell'ampia pagina riguardante i problemi, le peculiarità del gioco #2 sono ridotte rispetto al consueto gioco attivo. Molte strutture dati e funzioni, non sono più idonee e vanno modificate o addirittura eliminate.
Si pensi ad esempio alla funzione che si occupa della valutazione della qualità di una mossa. Diventa insignificante poiché l'unico interesse in questa modalità di gioco è sapere su uno spostamento porta ad un matto o meno. Oltre alla sola funzione di valutazione, decadono anche tutti gli elementi che contribuiscono alla scelta della mossa.
In TSCP gli spostamenti sono rappresentati mediante strutture dati in cui i campi non si limitano solo ad identificare il pezzo e le case di partenza e arrivo, ma contengono anche un campo che indica quanto beneficio tale spostamento può portare. Ebbene, anche questo campo non ha senso nel contesto dello scacco matto #2, ma ho preferito lasciarlo indenne ed ignorarlo per evitare ripercussioni sul resto del codice.
Anche la quiescenza non ha più ragion d'esistere.
Non è tutto qui: la modalità di funzionamento del programma subisce un pesante cambiamento. Si passa dal gioco interattivo ad un gioco “batch” dove l'utente da gli input alla macchina e attende i risultati dell'elaborazione. Scompaiono cicli utili al dialogo in fase di gioco, il menù delle ope-

razioni richiedibili dall'utente si riduce, mentre servirebbe una funzionalità in grado chiudere un problema ed aprirne un altro.

2.6 L'albero di ricerca

Il punto clou del calcolo dello scacco matto #2 ruota attorno alla funzione di ricerca. Questa, infatti, ha l'onere di visitare albero delle mosse legali e compiere precise operazioni su ogni nodo che sono ben differenti da quelle tipiche del TSCP proprio perché lo scopo non è quello di valutare la posizione ma di testare le situazioni di scacco matto dopo sole 3 semimosse. Il motore TSCP va profondamente modificato.
Prima di presentare il vero e proprio algoritmo, è utile fare qualche considerazione sull'albero di ricerca su cui si dovrà lavorare.

2.6.1 I livelli dell'albero

L'albero di ricerca è suddiviso in 4 livelli che indico in ordine decrescente per adeguarmi all'implementazione TSCP.

La radice è l'antenato di ogni nodo ed è unica. Essa posta a livello 4 e non è altro che la posizione iniziale dei pezzi. In altre parole, la radice contiene il problema non risolto.

Il livello 3 è composto dai figli della radice che corrispondono alla situazione sulla scacchiera dopo le possibili prime mosse del bianco. Questi rami rappresentano i movimenti che dovrebbero compromettere seriamente la situazione del re nero. Non si sa ancora quali di queste saranno chiave. Per ora sappiamo solo che se una qualche chiave esiste, allora la troviamo a questa profondità.
Se il problema è di pessima qualità, potrebbero esistere mosse in grado di mattare il nero. In tal caso il matto avviene con una mossa e il problema viene definito "demolito". Sarebbe inutile proseguire nell'esplorazione dell'albero a maggior profondità.

Ora che è avvenuta la prima mossa del bianco, il nero deve cercare di riparare alla situazione creatasi. Si cerca di sventare la minaccia di matto introdotta dal bianco mediante una mossa chiamata "difesa". Non è detto che la difesa sia necessariamente una mossa del re nero.
Già a livello 2 ci si aspetta una rilevante esplosione combinatoria poiché per ogni figlio della radice avremo tutte le possibile difese del nero.
Il livello 1 è l'insieme di tutte le seconde mosse bianche. Fra queste, affinché il problema possa avere delle soluzioni, dovrebbero esistere delle mosse in grado di dare scacco matto al re nero. Analogamente a quanto detto per il livello 2, anche il bianco deve generare tutte le possibili mosse in risposta ad ogni mossa del nemico. Il numero di nodi cresce notevolmente, poiché ogni difesa nera ha come figli tutti i possibili attacchi bianchi finali.

Il calcolo dello scacco matto in due mosse si conclude qui, ma dal punto di vista implementativo si rende necessaria un'altra serie di chiamate della funzione generatrice di mosse legali. Queste servono per verificare che il nero non sia più in grado di fermare lo scacco matto. Dopo la seconda

mossa del bianco, viene chiamata la funzione di generazione delle mosse legali e se nessuna di queste è in grado di annullare lo scacco, allora il problema ha soluzioni.

Partendo da questo ultimo nodo dobbiamo risalire l'albero in modo da scoprire quale possa essere stata la sequenza che ha portato al matto e quindi farne le dovute considerazioni.

2.6.2 AND e OR

Un problema di scacco matto #2 vuole determinare tutte le sequenze di due mosse che potrebbero portare ad uno scacco matto a partire da una determinata configurazione di pezzi sulla scacchiera. Esistono tuttavia delle regole che rendono il problema valido e dei parametri per indicarne quanto possa essere interessante. Tali peculiarità emergono osservando l'albero di mosse mostrato in figura 4.

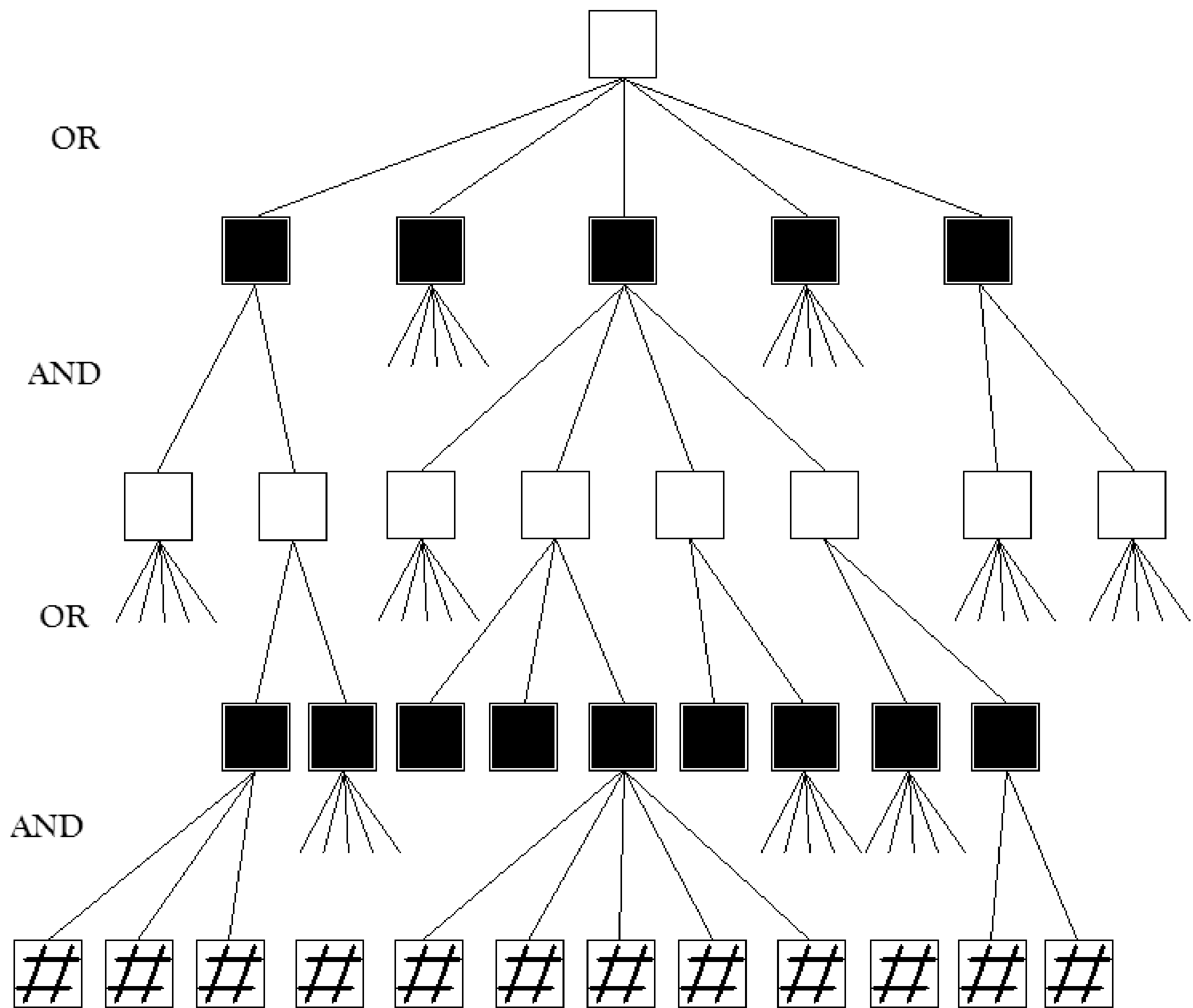


Figura 4: Albero e operatori logici

Purtroppo per motivi di spazio, non sono riuscito a rappresentare alcuni nodi, ma il ragionamento si ripete per qualunque nodo dell'albero entro le 3 semimosse previste.

Ad ogni nodo è associato un valore logico che dipende dal valore logico dei suoi figli. L'impiego di valori logici è dettato dall'esigenza di codificare due soli valori che corrispondono alla presenza o all'assenza di uno scacco matto proveniente dall'ultimo livello dell'albero. E' evidente che il controllo dello stato dei nodi avviene quando si risale la struttura dopo aver completato la generazione dei movimenti e dopo aver determinato se una sequenza di 3 semimosse è in grado di confermare il matto al re nero.

Nell'albero possiamo distinguere due operatori logici ed ognuno di questi è associato sempre allo stesso tipo di nodo (bianco o nero). Troviamo così l'operatore OR in corrispondenza dei nodi rappresentanti le mosse bianche e l'operatore AND in corrispondenza dei nodi rappresentanti le mosse nere.
Il fatto che ogni nodo sia il risultato di un'operazione logica calcolata sul valore dei figli consente la propagazione verso la radice degli eventi accaduti presso le foglie.
Probabilmente risulta più semplice presentare la situazione partendo proprio dalle foglie. E' proprio da qui che vengono generate le informazioni da ricondurre alla radice, poiché è a questo livel-

lo che avviene la determinazione della presenza di un matto.
Un matto è la situazione in cui il nero non riesce a trovare alcuna mossa in grado di parare lo scacco inflitto dal bianco. A questo livello dell'albero, il nero provvede a generare tutte le possibili mosse nella speranza di incontrarne almeno una che sappia evitare lo scacco. Quindi esamina tutte le mosse e ad ognuna assegna un valore che indica quel che succede se venisse eseguita.
Una mossa del nero che mantiene il re sotto scacco comporta la creazione di un nodo col valore logico TRUE. Una mossa nera in grado di impedire lo scacco verrebbe invece marcata col valore logico FALSE.
Se ad ogni nodo corrispondente alle mosse del nero associassimo uno di questi due valori logici, otterremo una serie di operandi su cui applicare la funzione AND. Il risultato ottenuto sarà il valore logico da assegnare al nodo padre.
L'operatore logico AND restituisce valore TRUE se tutti gli operandi hanno valore TRUE. Restituisce FALSE in qualunque altro caso. Se il nodo padre ha valore TRUE significa che qualunque mossa il nero compia, sicuramente non sarà in grado di evitare lo scacco e che quindi si è verificato un matto.

Il valore restituito da questo AND è l'elemento che certifica l'esistenza di un matto e risulta utile al livello superiore dell'albero per verificare che una mossa bianca sia in grado di realizzare l'enunciato del problema. A differenza del caso appena visto, il nodo padre ora è bianco ed è il risultato dell'operazione logica OR. Tutte le mosse che il bianco può effettuare in questo livello, rappresentano i modi con cui prova a dare scacco matto al nero. L'efficacia di ognuna di queste è indicata dai nodi figli visti poco fa. Applicare un OR logico fra i risultati ottenuti seguendo le mosse del bianco, significa determinare se esiste almeno una mossa in grado di mattare. Questo perché l'operazione OR restituisce TRUE se almeno uno dei suoi operandi ha valore TRUE. L'esistenza di più nodi figli con valore TRUE sono tuttavia una carenza del problema poiché ci sarebbero più modi di mattare per una medesima difesa (dualità).

Se il nodo padre bianco appena discusso assume valore FALSE, significa che qualunque mossa il bianco effettui, non porterebbe alcun beneficio, mentre la difesa nera si dimostrerebbe efficace. Il livello dell'albero che sto per chiarire è quello riferito alle difese nere e come visto per nodi di questo schieramento, il padre è il risultato di un'operazione di AND fra tutti i figli i quali contengono TRUE o FALSE a seconda dei risultati delle operazioni precedenti calcolate fino ad ora. Se il risultato dell'operazione fosse TRUE, significherebbe che qualunque difesa il nero effettui, incontrerà comunque uno scacco matto. Sicuramente tutti i nodi figli avranno valore TRUE. In caso contrario, un padre con valore FALSE indicherebbe la presenza di una qualche difesa in grado di evitare il matto.

Dopo diverse selezioni, si è arrivati finalmente al livello dell'albero che contiene i figli della radice. Fra questi figli dobbiamo prelevare solo quelli che hanno associato il valore logico TRUE, poiché solo quelli portano al matto indipendentemente dalla difesa che utilizzerà il nero. Anche qui troviamo il nodo padre bianco (la radice) che deve assumere un valore in base allo stato logico dei figli e anche qui dobbiamo applicare l'operatore OR. Il significato collegato è quello di stabilire se esistono soluzioni ossia se da qualche nodo figlio “emerge” uno scacco matto. Ovviamente la pre-

senza di più figli con valore TRUE non cambierebbe il valore della radice, ma indicherebbe la presenza di più chiavi nel problema. Avere una radice con valore FALSE, esprimerebbe l'assenza di soluzioni.

Detto questo, è stato presentato come l'algoritmo dovrebbe operare per trovare tutte le soluzioni del problema. Possiamo immaginare l'algoritmo come suddiviso in due fasi: la prima in cui si cerca di raggiungere i nodi foglia per determinare la presenza di un matto e la seconda in cui si risale l'albero eliminando man mano i rami che non portano ad alcuna soluzione.

2.6.3 Qualche numero

Ora che sono state espresse le caratteristiche che un albero per problemi di scacco matto #2, mi preme rendere l'idea di quanto questo albero possa essere grande. Mentre nei comuni software per il gioco degli scacchi, l'albero di ricerca può variare in profondità in base alla difficoltà scelta dall'utilizzatore, nel caso dello scacco matto #2 la profondità è ridotta e costante a meno che non si trovi un qualche matto in una sola mossa. Quindi ci si attende un albero più largo che profondo. Tuttavia anche la larghezza (numero di nodi ad uno stesso livello) dipende dai pezzi che entrano a far parte del problema.
A differenza di quanto si possa immaginare, il numero di figli che un nodo può avere non è necessariamente proporzionale al numero di pezzi presenti sulla scacchiera. E' risaputo infatti che ogni pezzo ha mobilità differente rispetto agli altri. Un pedone in posizione di partenza ha, nel migliore dei casi, la possibilità di spostarsi in 4 case (le due prese diagonali, l'avanzata di 1 casa e quella di due), mentre una torre può spaziare fra 14 case. Per finire la regina può raggiungere ben 27 case! Non è tutto qui: è rilevante anche il punto della scacchiera ove il pezzo giace. La posizione nei pressi di un bordo della scacchiera, comporta un minor numero di mosse effettuabili, mentre al centro della scacchiera si riesce a raggiungere il numero massimo di mosse possibili. Questa affermazione non è valida per i pedoni in settima traversa che avendo a disposizione 3 case per la promozione (le due catture diagonali e l'avanzata) possono raggiungere un totale di ben 12 mosse.
Tuttavia non dobbiamo dimenticare la presenza di altri pezzi (anche amici) che interferiscono le traiettorie di chi si muove lungo rettilinei. Una regina circondata da pezzi, non potrà esercitare tutta la sua copertura, mentre un alfiere libero al centro della scacchiera riuscirà a coprire fino a 13 case. Il cavallo invece è l'unico pezzo che si muove non seguendo rettilinei quindi non si pone problemi di interferenza con altri pezzi, se non con i quelli amici nella casa di destinazione.
Per concludere, ricordo che benché insolita, anche l'arrocco è una possibile mossa e quindi va aggiunta all'insieme dei nodi figli qualora sia disponibile.
E' chiaro quindi che due problemi simili, posso generare alberi di dimensioni molto diverse. A parità di profondità, un problema può essere molto più largo (molti rami figli) di un altro. Si pensi ad esempio ad a due problemi similari come i seguenti.

•*Esempio 1*

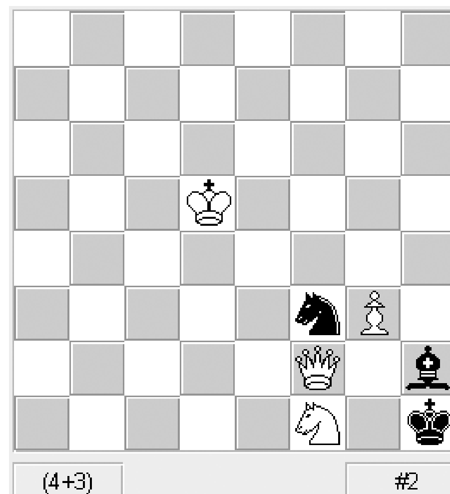


Figura 5: Josef Cumpe (Bohemia 1908)

Il programma trova la soluzione di questo problema esplorando ben 1916 nodi. Il tempo di esecuzione sulla mia macchina di casa supera di poco i 9 millisecondi.
Se si sostituisce la regina con una torre, non è detto che si riesca a trovare una soluzione (in effetti non v'è soluzione), ma otterremo una configurazione molto simile poiché la regina copre le stesse identiche case di una torre anche se ha facoltà di percorrere le linee diagonali. In questo caso, la regina riesce a coprire solo 7 case in più di una torre.
Lanciando il programma vediamo che non riesce a dare alcun matto.

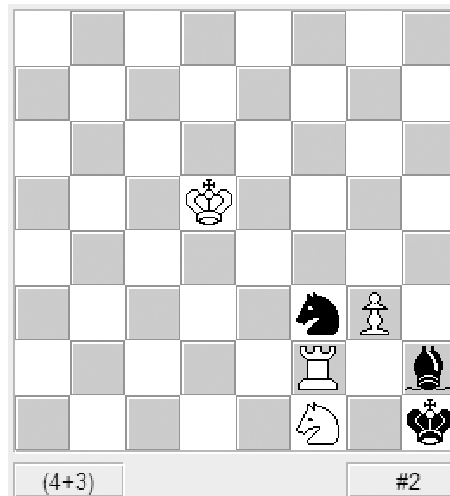


Figura 6: Una torre al posto di una regina

La soluzione quindi non esiste, ma il numero di nodi raggiunti scende vertiginosamente a 404.

•*Esempio 2*
Nel problema White Correction, rappresentato in figura 7, abbiamo come chiave Cb5 e minaccia Ca4. Ca4 non è la sola minaccia, poiché il compito del cavallo in b2 è quello di scoprire la regina portando lo scacco matto!

Figura 7: White Correction

La soluzione a tale problema è unica e avviene passando per ben 3314 nodi dell'albero.
Ora, potremmo pensare di sostituire il cavallo con un pedone, poiché anche lui ha facoltà di abbandonare la casa b2 per lasciar libera la minaccia della regina in d4.

Figura 8: Un pedone al posto del cavallo

Purtroppo così facendo, non otterremo alcuna soluzione poiché la difesa Tc7 metterebbe sotto scacco il re bianco. A differenza di prima, non esiste più il cavallo in grado di fermare tale minaccia (Cc4)senza oscurare la regina. Il numero di nodi visitati scende a 1636, la metà del caso precedente.

2.6.4 Tagli

Avere un minor numero di mosse possibili, comporta un albero di dimensioni inferiori. Vi sono tuttavia casi in cui non è necessario visitare tutti nodi dell'albero.
Nella maggior parte delle applicazioni si ha spesso a che fare con delle condizioni che se verificate permettono di ignorare parte della struttura dati poiché si ha la certezza che non contenga informazioni utili. Questo genere di operazione prende il nome di taglio. Si ha quindi un miglioramento delle performance che non comporta una soluzione migliore delle altre, ma permette di trovarle tutte in intervalli temporali visibilmente ridotti rispetto a quelli normalmente richiesti e con minor spreco di risorse. In molte circostanze ci si accontenta di una approssimazione, senza valutare l'albero fino in fondo.
Vediamo un esempio di come il programma può effettuare dei tagli.
L'immagine successiva dev'essere sufficientemente chiara per spigare la situazione.

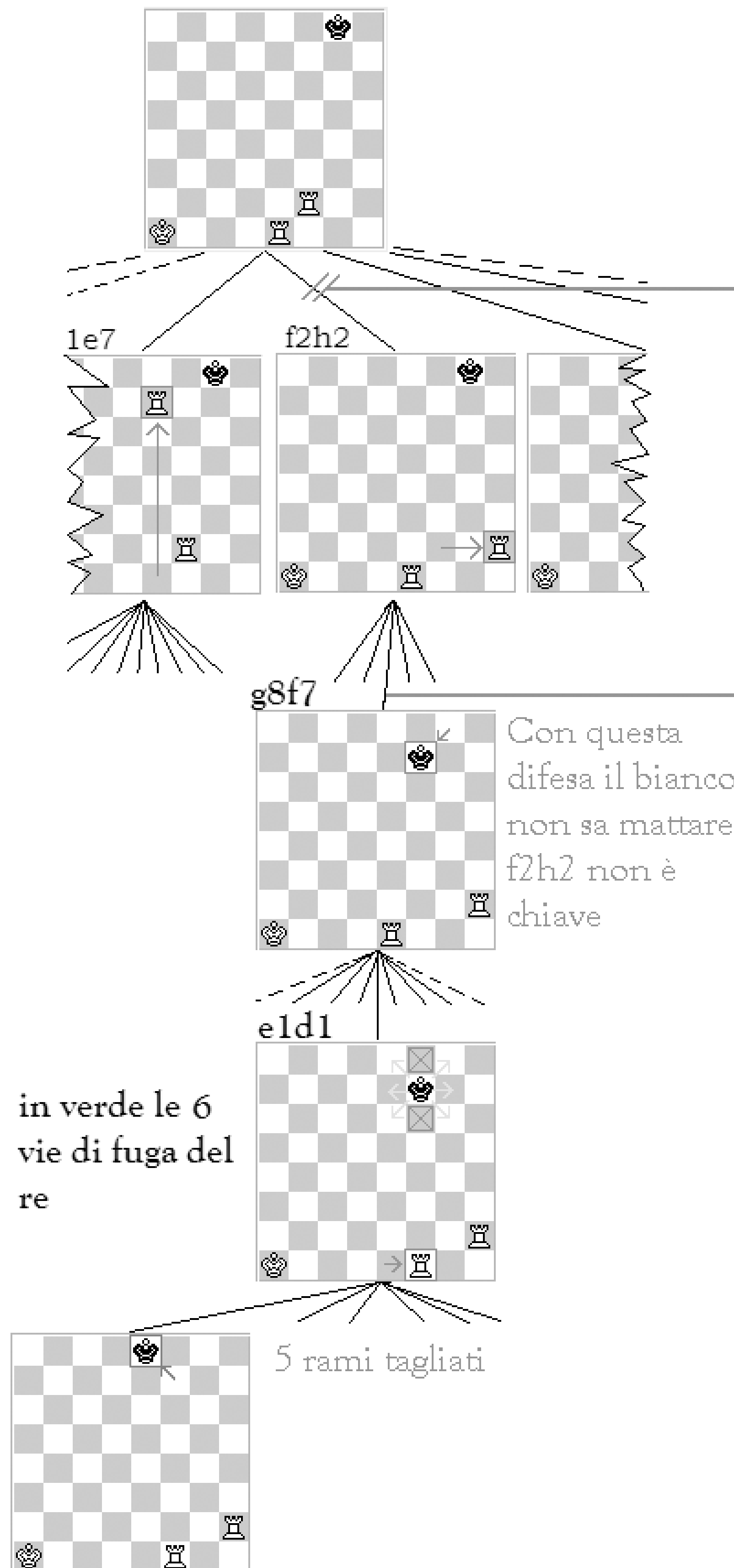


Figura 9: Tagli

In questo caso vediamo una prima mossa del bianco (f2h2) che tenta di compromettere la situazione del nero. Le difese nere sono composte da tre soli movimenti. Il re nero decide così di spostarsi verso la casa f7.
La successiva mossa del bianco è Tf1 e il re nero cade sotto scacco poiché minacciato dalla torre appena mossa. Se a questo punto si volesse calcolare se lo scacco è matto, dovremmo controllare che ogni possibile via di fuga del re sia minacciata dai pezzi bianchi. La condizione è verificata per le sole case f6 ed f8. Tuttavia, andrebbero controllate anche le altre aree copribili dal re. Una di queste è appunto la casa e8. Se il re nero fugge in questa casa, va sicuramente al riparo poiché nessun nemico è in grado di minacciarlo. Essendo questa mossa valida, appare ovvio che il bianco non è stato in grado di mattare il nero con la mossa Tf1. Dunque perché continuare con l'analisi delle altre casa raggiungibili dal re? Non è più necessario valutare cosa succede se il re nero fugge in g8 piuttosto che in e6. Quindi possiamo dedurre che per determinare se esiste uno scacco matto o meno, non è obbligatorio controllare tutte le possibilità. In questo caso siamo riusciti a risparmiare la copertura di soli 5 nodi, ma si pensi a quanti se ne potrebbero risparmiare se ci fossero anche altri pezzi sulla scacchiera.

Ancor più interessante è analizzare come le regole dei problemi consentono di evitare l'analisi di nodi inutili. Richiamo l'attenzione allo schema AND OR visto nel paragrafo precedente (figura 4), che reputo essere il nocciolo di tutta la problemistica.

Partendo dalle foglie dell'albero troviamo potenziali tagli dovuti alla condizione di scacco matto. Non intendo approfondire questo punto poiché già stato trattato largamente poco fa. Mi limito semplicemente a dire che se viene trovata una sola via di fuga in grado di evitare lo scacco matto, si può rinunciare al controllo di tutte le possibili mosse nere.

Al livello superiore troviamo le mosse bianche che tentano di dare matto al nero. Se il nostro scopo fosse semplicemente quello di verificare che esista una mossa in grado di chiudere la partita, allora potremmo operare dei tagli qualora si trovi un matto. Al contrario, invece, dobbiamo determinare tutti i modi possibili per dare matto e quindi, a questo livello, non è possibile effettuare tagli.

Poco più su ci sono le difese nere che introducono le diverse varianti del gioco. Affinché una chiave possa essere tale, è indispensabile che ogni difesa del nero possa essere respinta da un matto del bianco. Se dai livelli sottostanti si determina l'assenza di un matto, significa che non tutte le difese si dimostrano sconfitte. Teoricamente è inutile analizzare le successive difese del nero, poiché se n'è già trovata una in grado di annullare la chiave. E' inutile proseguire nella ricerca di nuovi matti fra le varianti della medesima chiave, poiché essa non è più tale e si può procedere con l'analisi di un'altra prima mossa bianca.
E' proprio questo il caso del secondo taglio rappresentato nell'esempio. Passando in rassegna tutte le seconde mosse del bianco dopo la difesa Rf7, ci si accorge che non si riesce in alcun modo a dare scacco matto. Ciò implica che la prima mossa bianca non è sufficientemente aggressiva poiché lascia al nero una difesa in grado da evitare il pericolo. Possiamo quindi terminare l'analisi di tutte le difese nere poiché ne è già stata trovata una in grado di annullare la chiave.

Per quanto riguarda i figli della radice, non possiamo operare alcun taglio a meno che non ci si limiti a determinare l'unicità della chiave. Al compositore invece interessa sapere se esistono e quali sono tutte le chiavi, affinché le possa ridurre ad una per aumentare la qualità del problema.

2.7 L'algoritmo

Ora abbiamo tutti gli elementi di analisi per poter stendere l'algoritmo di ricerca di soluzioni nei problemi di scacco matto #2. Vediamo brevemente come dovrebbe essere lo pseudocodice della funzione di ricerca. Successivamente verrà presentato come ho implementato questo codice in linguaggio C all'interno dell'applicativo.

La funzione di generazione dell'albero e di ricerca del matto in due mosse, segue una logica molto semplice, benchè richieda innumerevoli chiamate ricorsive.
Una volta determinata la radice, ovvero il problema di partenza, l'algoritmo genera tutte le mosse bianche e ne esegue una ad una. Per ogni mossa effettuata, vengono generate tutte le difese nere e per ognuna di queste vengono generate di nuovo le mosse bianche. A questo punto, avviene un'ultima generazione di mosse nere, ma con lo scopo di osservare se ne esistono alcune in grado di sventare un eventuale scacco causato dal bianco. Dopo quest'ultimo passo, inizia l'ascesa dell'albero che propaga gli esiti verso la radice. In questa fase, l'algoritmo tiene conto delle condizioni di gioco della problemistica mediante le combinazioni dei valori di ritorno delle funzioni.

L'esecuzione dell'algoritmo di ricerca deve far fronte a sei possibili situazioni che si possono verificare dopo aver effettuato una mossa, in modo da poter stabilire che azione intraprendere. Le situazioni possibili sono le sei elencate quì sotto e sono identiche a quelle del comune gioco attivo. Eccole:

1- il bianco ha mosse valide ed esiste un matto
2- il bianco ha mosse valide e non esiste matto
3- il nero ha mosse valide ed esiste un matto
4- il nero ha mosse valide e non esiste matto
5- non esistono mosse valide quindi c'è un matto
6- non esistono mosse valide quindi c'è uno stallo

Ognuno di questi casi è mutuamente esclusivo. La funzione di ricerca deve garantire che tutte le situazioni vengano coperte.
La funzione di ricerca è suddivisa in tre parti: un ciclo e due istruzioni di controllo.

Ognuna di queste parti gestisce due dei casi citati e in particolare:
- Il primo blocco costituito dal ciclo gestisce i casi 1 e 4
- Il secondo blocco gestisce i casi 5 e 6
- Il terzo blocco gestisce i casi rimanenti ovvero 2 e 3

In breve presento lo pseudocodice della funzione di ricerca.

```
funzione RICERCA( profondità) ritorna situazione
inizio RICERCA
        se profondità = 0
                ritorna "non matto"

        controlla se c'è scacco e salvalo in una variabile

        genera tutte le mosse di chi tocca
-----------------------------------------------------------
        inizia ciclo
                esegui una mossa di quelle generate
                        se non è possibile eseguirla passa alla successiva

                x = RICERCA(profondità -1)

                annulla la mossa eseguita

                se tocca al bianco
                        se c'è un matto
                                se profondità è 4
                                        salva la chiave e continua col ciclo
                                altrimenti
                                        ritorna il matto al chiamante
                altrimenti
                        se c'è matto
                                continua col ciclo
                        altrimenti
                                ritorna "non matto" al chiamante
        fine ciclo
----------------------------------------------------------
        (Giunti quì significa che non siamo riusciti ad eseguire nessuna mossa nel ciclo)
                se c'è scacco
                        se profondità =3
                                è un matto in uno
                        altrimenti
                                salviamo il matto
                        comunque ritorniamo matto al chiamante
                altrimenti
                        ritorna "non matto" al chiamante
-----------------------------------------------------------
```

(se sono giunto quì, significa che ho eseguito mosse ma non ho trovato uscita)
se tocca al bianco
non ha saputo dare matto, ritorna "non matto"
altrimenti
il nero ha mosse valide ma trova sempre matto, ritorna matto al chiamante
----
fine RICERCA

*2.7.1 **Search()***

La funzione search() riprende pari pari lo pseudocodice appena visto ma si deve attenere alle parole chiave del linguaggio di programmazione impiegato.
Riporto ora l'intera funzione search() aggiungendo una descrizione delle istruzioni fondamentali.

```
BOOL search( int depth)
{
     int i;
     BOOL x,c, f;

     if (!depth){
          return FALSE;
     }
```

La funzione search() è la funzione che genera e visita l'albero in ogni suo nodo. Search() riceve in ingresso il livello a cui opera e restituisce un booleano che indica la situazione nel nodo corrente e di conseguenza anche nei nodi sottostanti, rispettando le condizioni logiche viste nel paragrafo 2.6.2 .

Dopo aver dichiarato le variabili locali necessarie alla funzione viene immediatamente fatto un controllo che permette di valutare a che livello dell'albero si è giunti. La scrittura "if(!depth)" è analoga a "if(depth==0)" e sta a significare che il blocco di codice sottostante deve essere eseguito solo se si è giunti a livello 0 ossia quello delle foglie. Quindi se la variabile "depth" ha valore nullo, allora significa che abbiamo raggiunto una foglia dell'albero senza aver trovato alcun matto. In tal caso non ci resta altro che uscire dalla funzione e notificare l'accaduto al chiamante ritornando il valore logico FALSE. Qualora invece "depth" abbia un valore diverso da 0, si procederà con l'esecuzione del codice riportato oltre questa istruzione di controllo.
Ricordo che TSCP indica i livelli dell'albero dal maggiore al minore, partendo dalla radice.

```
     ++nodes;
```

L'istruzione appena scritta, non è rilevante ai fini del calcolo dello scacco matto in due mosse, ma è interessante dal punto di vista statistico. La variabile globale "nodes" non è altro che un contatore di nodi visitati e ne viene incrementato il valore ogni volta che si accede ad un nuovo nodo dell'albero. TSCP incorpora funzioni che sfruttano questa variabile per determinare quanti nodi vengono visitati nell'unità di tempo.

```
c = in_check(side);
```

La funzione in_check() è in grado di stabilire se il re dello schieramento passato come argomento è sotto scacco. Il parametro attuale "side" è una variabile booleana che può contenere due soli valori. Quando "side" è uguale ad 1, significa che il tratto è nero, mentre quando "side" è 0, il tratto è bianco. Passare come parametro "side" alla funzione in_check(), significa richiedere la valutazione delle posizioni sulla scacchiera per determinare se lo schieramento in questione è sotto scacco. Il valore restituito è ovviamente booleano. Se c'è scacco al re dello schieramento "side", alla variabile "c" verrà assegnato valore TRUE, altrimenti verrà assegnato valore FALSE. In seguito ne vedremo l'utilizzo.

```
gen();
```

gen() è una sofisticata funzione che permette la generazione tutte le mosse legali di tutti i pezzi di uno stesso colore. Prima di procedere con la generazione, viene testato il valore di "side" , indispensabile per capire di quale schieramento bisogna creare le mosse. gen() provvede a salvare tutti i movimenti legali in un'apposita struttura dati (gen_dat[]) che verrà poi utilizzata per creare nuovi nodi figli.

```
f = FALSE;
for (i = first_move[ply]; i < first_move[ply + 1]; ++i){
     if (!makemove(gen_dat[i].m.b))
          continue;
     f = TRUE;
```

La prima di queste istruzioni è una condizione iniziale ed indica l'assenza di mosse legali.
Il ciclo si occupa di prendere in considerazione tutte e sole le mosse generate dopo l'ultima chiamata di gen() e di eseguirle. Il numero di iterazioni che il ciclo deve compiere viene estratto dal vettore first_move[] il cui compito è di memorizzare il range di indici del vettore gen_dat[] in cui possiamo trovare le mosse da realizzare. Il contatore "i" assume tutti i valori del range in modo da richiedere l'esecuzione di tutte le mosse da poco salvate nel vettore ad opera della gen().
Per determinare se la mossa prelevata è eseguibile nell'attuale contesto, viene chiamata la apposita funzione makemove(). Essa compie realmente uno spostamento sulla scacchiera e ne valuta gli effetti. Terminato lo spostamento, makemove() richiama la funzione in_check() ma stavolta per verificare che la mossa appena eseguita non permetta alcuna minaccia al proprio re. Se così fosse, non sarebbe valida e verrebbe immediatamente annullata. Ad una mossa legale non verrà invece applicata nessun'altra modifica e la funzione makemove() ritorna TRUE poichè l'operazione è andata a buon termine. Grazie a questo secondo caso, la variabile "f" potrà assumere valore "true", essenziale per stabilire il flusso di esecuzione nel resto della funzione.

Avere un consistente numero di mosse legali da eseguire, determina un maggior numero di nodi figli che comporta l'aumento della larghezza dell'albero stesso.

```
x = search( depth - 1);
```

La creazione dei nodi figli avviene grazie alla combinazione del ciclo visto poco fa con la presente search(). Per ogni mossa eseguibile, viene chiamata una search() in modo ricorsivo. Il fatto che i nodi generati siano a livello sottostante, è motivato dal parametro passato, ossia "depth" decrementato. Per come è impostata tale funzione, ne deriva la visita dell'albero in profondità (Depth-First Search - DFS), vale a dire un'esplorazione in cui si procede dalla radice percorrendo i rami sino alle foglie e ritornando indietro all'ultima diramazione non esplorata, per proseguire allo stesso modo la visita dei vertici sugli altri rami.

```
takeback();
```

takeback() è una funzione analoga a makemove(), solo che compie l'esatto contrario, ossia annulla la mossa eseguita. Viene utilizzata quando si "risale" l'albero per ripristinare la situazione precedente alla mossa consentendo l'analisi dei branch successivi. E' proprio da questo punto che dobbiamo applicare le regole dei problemi di scacco matto #2. Quello che stiamo per affrontare, unito alle due parti successive, è il nucleo dell'algoritmo per il calcolo dello scacco matto diretto in due mosse.

```
        if(!side){
             if (x ){            //c'è un matto
                  if(depth==4) {

                  chiavi[contachiavi++].m.b=hist_dat[0].m.b;
                  }
                  else
                       return x;
             }
        }else{
             if (x ){

             }else{
                  return x; //il nero ha trovato vie di fuga
             }
        }
   }
```

La prima condizione, serve per determinare il comportamento dell'algoritmo nel caso il tratto sia bianco piuttosto che nero. Se viene verificato "if(!side)" significa che il turno è bianco.
Compito del bianco, indipendentemente dal livello dell'albero, è solo quello di notificare al padre se seguendo quel ramo si raggiunge un matto. E' quindi chiaro che se la variabile "x" ha valore

TRUE, la funzione search() debba ritornare al padre proprio TRUE. Va tuttavia segnalata un'eccezione. Quando si opera a livello 4, non è possibile restituire al chiamante alcun valore, poichè si interromperebbe l'algoritmo di ricerca. La soluzione trovata è stata quella di memorizzare la chiave in un'apposita struttura dati e di proseguire con ricerche successive, in attesa della completa analisi di tutte le potenziali chiavi del problema. Se viene verificata la condizione "if(depth==4)" l'algoritmo copia la mossa alla base dello stack delle mosse eseguite (che coincide con la prima mossa bianca) in un vettore contenente tutte e sole le chiavi del problema.

Nel caso del nero, la situazione si complica, poichè bisogna fare una valutazione di tutti i figli di uno stesso nodo. Non sarà sufficiente osservare singolarmente il valore di ritorno dei nodi sottostanti per generare un valore d'uscita dalla search(). Se dopo la mossa nera, si scopre di aver ricevuto un matto dal bianco, non significa che il nero non abbia a disposizione altre vie di fuga o mosse in grado di sventare il matto bianco. Quindi, per ora, non possiamo ritornare alcun valore, poichè dovremmo testare tutte le mosse del nero, nella speranza che se ne trovi una a cui il bianco non sia in grado di mattare. Si pensi al consueto schema ad albero ed alle condizioni di AND sulle mosse nere.

Purtroppo non abbiamo ancora coperto tutti i casi che si possono verificare. Esiste infatti la situazione in cui uno schieramento non abbia mosse legali. Se così fosse, significherebbe che la variabile "f" non è mai stata modificata poichè la makemove() non è mai andata a buon termine ed ha sempre ritornato "false".

```
    if (!f) {
        if (c){
                matti[contatore][0].m=hist_dat[0].m;
                if(depth==3){          //se vero, allora è
                                       //matto in 1
                    contatore++;
                }else{
                    matti[contatore][1].m=hist_dat[1].m;
                    matti[contatore++][2].m=hist_dat[2].m;
                }

            return TRUE;
        }
        else{
            if(!side){         //il bianco non ha mosse valide e
                               //non è sotto scacco
                    //soluzione non di interesse perchè non
                    //porta a matto
            return FALSE;
            }else{             //il nero non ha mosse valide ma
                               //non è in scacco
                    //anche qui la soluzione non porterebbe
                    //a matto
                    //anzi, porterebbe ad uno stallo!
```

```
                return FALSE;
                }
        }
}
```

Se non esistono mosse valide e si è verificato uno scacco, allora c'è un matto ed è bene che venga memorizzato in una struttura dati apposita. E' in questo punto che viene utilizzata la variabile "c" inizializzata tra le prime righe della funzione. Purtroppo esistono casi in cui il matto viene generato a livello 3. In tali situazioni il problema non è corretto poichè ammette matti in una mossa. Comunque sia, la return notifica al livello superiore la presenza di un matto, anche se attua una gestione delle strutture dati differente. Leggendo il codice si capisce che i dati salvati nel vettore matti[] sono rispettivamente chiave, difesa e matto. Tutte queste informazioni vengono trasferite nel vettore matti[]. Nel caso del matto in una mossa, la chiave e il matto coincidono, mentre la difesa non esiste. Per questo motivo, verrà copiata la sola chiave nel vettore matti[].
Il caso complementare al precedente indica le situazioni in cui sia il bianco che il nero non hanno mosse valide, ma i loro re non sono minacciati da alcun pezzo nemico. Nel caso bianco, possiamo tranquillamente restituire FALSE al chiamante poichè anche qui non si verifica alcun matto.
La medesima condizione per il nero significa stallo e possiamo ritornare FALSE.

```
if(!side)
        return FALSE;
else{
                //se sono nero e ho mosse valide e ho tro
                //vato sempre matto,allora restituisci matto
        return TRUE;
}
```

Raggiungere questa sezione di codice è la conseguenza di una serie di iterazioni nel ciclo che hanno si generato mosse valide, ma mai nessuna di queste ha portato ad una qualche return. Ovviamente, la presenza di mosse valide non ha permesso di soddisfare la condizione "if(!f)" del blocco precedente.

Il caso bianco del''istruzione di controllo significa che a fronte delle diverse mosse valide non si è stati in grado di dare scacco matto al nero. Per questo motivo viene restituito il valore FALSE.
Al contrario, nel caso nero ci si trova nella situazione in cui nessuna mossa permette di evitare le minacce che il bianco impone. La funzione ritorna TRUE.

Il calcolo dello scacco matto in due mosse termina qui, poiché con il presente algoritmo sono state trovate tutte le soluzioni del problema. L'ampia finestra che sto per aprire tratta la ricerca delle numerose informazioni che ancora possono essere estratte dal problema. Tutti questi elementi consentono al compositore di tracciare un profilo generale del problema e di arricchire il contenuto informativo ad esso associato.

2.8 Minaccia e blocco

I problemi di scacco matto #2 si suddividono in due grandi categorie: i problemi a minaccia ed i

problemi a blocco. In entrambi i casi l'obiettivo rimane lo stesso, ma ognuno gode di particolari significativi che rendono il problema interessante. Vediamo ora di che si tratta e come questi tipi di problemi vengono riconosciuti nel programma.

2.8.1 Minaccia

Un problema di scacco matto #2 viene definito " a minaccia " se, una volta effettuata la mossa chiave, il bianco ha a disposizione una seconda mossa in grado di dare matto senza che il nero muova. In particolare, questa seconda mossa è la **minaccia**. La minaccia non è necessariamente unica, ve ne possono essere diverse, a tal punto che vi sono delle competizioni per compositori fondate sull'esistenza e ricerca di tutte le possibili minacce (Tema Fleck).
Una minaccia non è necessariamente la mossa mattante del problema, anzi, la maggior parte delle volte non coincide affatto. Viene tuttavia esplicitata per fornire maggiori dettagli del problema.

2.8.1.1 Il codice

Come si è intuito, trovare una minaccia non permette di trovare le soluzioni del problema quindi non è un'operazione eseguibile in parallelo a meno che non si complichi la funzione dei scansione dell'albero search(). Per semplicità ho preferito separare gli obiettivi ed agire in tempi diversi, presentando comunque gli output simultaneamente in modo da semplificare la comprensione.
Trovare una minaccia, ha una sola differenza rispetto alla ricerca del consueto matto in due mosse, ossia il nero non attua difese. Ciò significa che il bianco valuta per due volte consecutivamente le posizioni sulla scacchiera.
Così ho riscritto la funzione di ricerca preesistente decurtandola delle parti inutili e vietando qualsiasi mossa nera che non sia per il controllo dello scacco matto.
Una volta memorizzate tutte le sequenze mattanti (le 2 mosse bianche), dobbiamo assicurarci che le ipotetiche chiavi siano le stesse del problema, altrimenti le minacce non sarebbero tali.
E' proprio per questo motivo che si è reso necessario calcolare prima tutte le soluzioni e poi tutte le minacce.

Se si pensa alla solita struttura ad albero, ci si rende conto che ignorare le mosse del nero consiste nel "saltare" il livello 3 dell'albero, ossia quello in cui risiedono le difese. Bisogna inoltre porre particolare attenzione alla gestione del turno per non generare mosse dello schieramento sbagliato. Riporto un frammento di codice per chiarire meglio la situazione.

```
BOOL TrovaMinaccia(int prof)
{
     int i;
     BOOL x;
     BOOL c, f;

     if (!prof)   return FALSE;
```

In queste prime righe viene effettuato lo stesso controllo visto per la funzione search() che determina il livello dell'albero raggiunto.

```
if(prof==2)
          c = in_check(!side);
     else
          c = in_check(side);
if(c&&(prof==2)){
          return FALSE;
}
```

E' evidente che se abbiamo raggiunto il livello 2 dell'albero e si è verificata una condizione di scacco, allora non esiste alcuna minaccia poichè la prima mossa ha già messo sotto scacco il re nero, quindi viene ritornato valore FALSE alla funzione chiamante.

```
     gen();
     f = FALSE;
     for (i = first_move[ply]; i < first_move[ply + 1]; ++i){
          if (!makemove(gen_dat[i].m.b))
               continue;
          f = TRUE;

          if(prof==4){
               side^=1;
               xside^=1;
               x = TrovaMinaccia(prof - 2);
               side^=1;
               xside^=1;
          }
          else
               x=TrovaMinaccia(prof-1);
...
```

Il funzionamento di TrovaMinaccia() è sostanzialmente identico a search() visto che hanno lo stesso compito, ma con una lieve differenza. Nel caso di TrovaMinaccia(), dobbiamo gestire i turni in modo differente non permettendo che il nero giochi immediatamente dopo la prima mossa del bianco. Per questo motivo, viene chiamata la funzione TrovaMinaccia() passando una profondità decrementata di due unità se deve venire eseguita la seconda mossa del bianco. Anche la gestione dei turni è inconsueta: per poter riutilizzare il codice già esistente ho dovuto alterare il turno corrente prima e dopo la chiamata della TrovaMinaccia(prof-2) .
Per quanto riguarda gli altri livelli, rimane tutto tale e quale e la chiamata ricorsiva passa come parametro la profondità decrementata come sempre di uno.
La gestione dei valori ritornati non ha richiesto alcuna modifica ed è stata ripetuta pari pari.

Ora, ogni potenziale minaccia è stata memorizzata in una matrice. Prima di stampare a video gli output ottenuti, dobbiamo fare un matching fra le potenziali chiavi e le chiavi del problema. Ecco come ho risolto il problema:

```
void StampaMinacce(hist_t chiave){
int p;
for (p=0;p<contaminacce;p++){

if((chiave.m.b.from==minacce[p][0].m.b.from)&&(chiave.m.b.to==
minacce[p][0].m.b.to)){
     printf(" [");
     printf(move_str(minacce[p][1].m.b));printf("#] ");
     mina=1;
     }
}
```

Naturalmente la funzione StampaMinacce() deve essere chiamata una volta sola ma per tutte le chiavi presenti nel problema. Per questo motivo la funzione richiede come parametro in ingresso, la chiave di riferimento.
Ed ecco un possibile output di un problema con più minacce:

```
C:\TESI\TESI.exe
benvenuti!

PROGRAMMA A SUPPORTO DEI COMPOSITORI DI PROBLEMI SCACCHISTICI
INSERISCI IL NOME DEL FILE:whitecorrection.fen
APRO IL FILE: whitecorrection.fen

8   . . . . . . . .
7   . . . r . . . .
6   . . . . . . . .
5   . . . P . . . .
4   . p . Q . . . .
3   . . N . . . . .
2   . N . . . . . .
1   k B K . . . . .

    a b c d e f g h


’on’ per avviare
’on-all’ per avviare e mostrare tutte le difese
’d’ per mostrare la scacchiera
’new’ per cambiare problema
’bye’ per uscire
>on
-------------------------------------------------
ECCO TUTTE LE SOLUZIONI:
SOLUZIONE 1:c3b5

CHIAVE: c3b5 [b2a4#]  [b2c4#]  [b2d3#]  [b2d1#]
        VARIANTI:
        d7c7    -->     b2c4#
        d7d5    -->     d4a7#
        b4b3    -->     d4a4#
-------------------------------------------------
Esaminati 3348 nodi in 4.000000 millisecondi

’on’ per avviare
’on-all’ per avviare e mostrare tutte le difese
’d’ per mostrare la scacchiera
’new’ per cambiare problema
’bye’ per uscire
>_
```

Figura 10: Problema con più minacce

*2.8.2 **Blocco***

La seconda categoria è composta da tutti quei problemi che non hanno alcuna minaccia. Il che significa che se il bianco effettua la chiave, non riesce a trovare una mossa mattante a meno che il nero non muova e quindi peggiori la sua situazione. Quindi, un problema in cui il bianco gioca la prima mossa che costringe il nero a commettere errori che peggiorano la sua situazione (zugzwang) agevolando il matto è chiamato problema “ a blocco ”.

Figura 11: Complete Block

In questo problema denominato Complete Block, il bianco non deve far altro che trovare una mossa che permetta di attendere l'errore del nero per poter mattare. L'unica mossa bianca che permette questa “attesa” è Rg6 poiché qualsiasi altro movimento (Re7, Re6, Rf5, Rg5)concede al nero di inchiodare dei pezzi bianchi.
Come si può vedere dall'immagine, il re nero non può fuggire in nessuna casa e tutti gli altri pezzi hanno mobilità ridotta.

```
C:\TESI\TESI.exe
benvenuti!

PROGRAMMA A SUPPORTO DEI COMPOSITORI DI PROBLEMI SCACCHISTICI
INSERISCI IL NOME DEL FILE:completeblock.fen
APRO IL FILE: completeblock.fen

8  . . . . . . . .
7  . . N . . . . .
6  . p N . . K . .
5  r . p Q . p . .
4  p . . . . B . .
3  p . k p . . . .
2  P p b R . . . .
1  . n b . . . . .

   a b c d e f g h


'on' per avviare
'on-all' per avviare e mostrare tutte le difese
'd' per mostrare la scacchiera
'new' per cambiare problema
'bye' per uscire
>on
-------------------------------------------------
ECCO TUTTE LE SOLUZIONI:
SOLUZIONE 1:f6g6

CHIAVE: f6g6 [BLOCCO]
        VARIANTI:
        b6b5    -->     d5c5#
        a5a6    -->     c7b5#
        a5a7    -->     c7b5#
        a5a8    -->     c7b5#
        a5b5    -->     c7b5#
        c5c4    -->     d5d4#
        c2b3    -->     d5d3#
        c2d1    -->     d5d3#
        b1d2    -->     f4e5#
        c1d2    -->     f4e5#
-------------------------------------------------
Esaminati 5413 nodi in 10.100000 millisecondi
```

Figura 12: Output di un problema a blocco

Fra le varianti appaiono tutte e 10 le possibili mosse dei pezzi neri, ma a tutte il bianco sa rispondere con un matto.

2.9 Difese efficaci

Dopo numerosi test, ho notato che il programma rispondeva correttamente a tutti i problemi ma generava anche delle soluzioni "poco interessanti" benché rispondessero alla definizione di soluzione. Che cosa significava?

Il programma prova tutte le possibili difese del nero ed ad ognuna di queste cerca i possibili matti. Fra le difese del nero si trovano anche quelle mosse che non apportano un cambiamento significativo del problema e verrebbero generate molte varianti simili. Si prenda ad esempio il seguente problema:

Figura 13: White Correction

Questo problema ammette ben 11 varianti di cui 6 hanno lo stesso matto!

```
C:\TESI\TESI.exe
------------------------------------------------
ECCO TUTTE LE SOLUZIONI:
SOLUZIONE 1:c3b5

CHIAVE: c3b5 [b2a4#]  [b2c4#]  [b2d3#]  [b2d1#]
        VARIANTI:
        d7d8    -->     d4a7#
        d7c7    -->     b2c4#
        d7b7    -->     b2a4#
        d7a7    -->     d4a7#
        d7e7    -->     b2a4#
        d7f7    -->     b2a4#
        d7g7    -->     b2a4#
        d7h7    -->     b2a4#
        d7d6    -->     b2a4#
        d7d5    -->     d4a7#
        b4b3    -->     d4a4#
------------------------------------------------
Esaminati 3314 nodi in 6.000000 millisecondi

'on' per avviare
'on-all' per avviare e mostrare tutte le difese
'd' per mostrare la scacchiera
'new' per cambiare problema
'bye' per uscire
>_
```

Figura 14: Output White Correction

Se si osservano attentamente le difese del nero, troviamo tutte le mosse della torre nera in d7 più l'avanzata del pedone (il re non può muovere). E' chiaro che difese come Tf7 o Td6, non hanno alcun significato poiché non complicano in alcun modo la situazione al bianco. Più interessanti potrebbero essere invece difese come Tc7 o Td5. Ebbene, queste difese unite a b3, sono le uniche in grado di sventare almeno una delle minacce sopraelencate!
Infatti, dopo la chiave Cb5, la difesa Tc7 costringerebbe il bianco a mettere il proprio re al riparo dalla minaccia della torre e impedirebbe il matto.

Se la torre nera mangiasse il pedone in d5, verrebbe vanificata la mossa del cavallo in b2. Infatti la regina non potrebbe più minacciare il re poiché potrebbe essere catturata dalla torre che anche in questo caso impedisce il matto.
Infine, il pedone nero tenta di interferire lungo la linea d'azione della regina, contrastando la minaccia.

Ho quindi ritenuto opportuno rimettere le mani sul codice e cercare di rendere all'utente solo le informazioni salienti. Ovviamente si poteva giocare anche sulle combinazioni di minacce, ossia presentare le sole difese in grado sventarne una, oppure due oppure tutte. Per convenzione ho preferito mostrare tutte le difese in grado di sventare almeno una minaccia. L'output sarà:

```
C:\TESI\TESI.exe
benvenuti!

PROGRAMMA A SUPPORTO DEI COMPOSITORI DI PROBLEMI SCACCHISTICI
INSERISCI IL NOME DEL FILE:whitecorrection.fen
APRO IL FILE: whitecorrection.fen

8  . . . . . . . .
7  . . . r . . . .
6  . . . . . . . .
5  . . . P . . . .
4  . p . Q . . . .
3  . . N . . . . .
2  . N . . . . . .
1  k B K . . . . .

   a b c d e f g h

'on' per avviare
'on-all' per avviare e mostrare tutte le difese
'd' per mostrare la scacchiera
'new' per cambiare problema
'bye' per uscire
>on
--------------------------------------------------
ECCO TUTTE LE SOLUZIONI:
SOLUZIONE 1:c3b5

CHIAVE: c3b5 [b2a4#] [b2c4#] [b2d3#] [b2d1#]
        VARIANTI:
        d7c7    -->     b2c4#
        d7d5    -->     d4a7#
        b4b3    -->     d4a4#
--------------------------------------------------
Esaminati 3348 nodi in 4.000000 millisecondi

'on' per avviare
'on-all' per avviare e mostrare tutte le difese
'd' per mostrare la scacchiera
'new' per cambiare problema
'bye' per uscire
>_
```

Il problema ha 4 minacce. Le difese presentate sono quelle che ne sanno sventare almeno una

Figura 15: Output White Correction con le sole difese efficaci

La selezione delle difese efficaci, non sarebbe stata così banale se non avessi potuto usufruire delle strutture dati che ho creato nelle fasi precedenti del calcolo dello scacco matto in 2 mosse. Avevo organizzato 2 matrici: una contenente tutte i possibili matti ed una contenente tutte le sequenze di due mosse bianche in grado di mattare. Ricordo che una minaccia, per essere tale, deve avere al prima mossa identica alla chiave del problema.
Per determinare se una difesa è in grado di sventare una minaccia, bisogna agire nel seguente modo:

1. Eseguire la chiave
2. Eseguire la difesa
3. Tentare di eseguire ogni minaccia
4. Verificare se esiste matto

E' possibile che a seguito della difesa, la minaccia non sia più realizzabile. In tal caso possiamo dire che la difesa si è veramente dimostrata efficace. Ciononostante, non è detto che se anche la minaccia venisse eseguita, la difesa si sia dimostrata ininfluente. Si pensi ad esempio ad una difesa che si frappone fra il re ed il pezzo minacciante. L'assenza di scacco matto dopo l'esecuzione della minaccia garantisce l'efficacia della difesa.
Vediamo come è stata implementata questa situazione:

```
BOOL efficace(hist_t  key, hist_t defence){
```

La funzione riceve in ingresso la chiave e la difesa da esaminare. Tutte le altre informazioni necessarie sono contenute in variabili globali e non serve passarle come parametri in ingresso.

```
    int i;int ics;
    makemove(key.m.b);
    makemove(defence.m.b);
```

Una volta dichiarate le poche variabili, vengono eseguite rispettivamente la chiave e la difesa.

```
    for(i=0;i<100;i++){
        if((key.m.b.from==minacce[i][0].m.b.from)&&
        (key.m.b.to==minacce[i][0].m.b.to)){
```

Questo ciclo permette il confronto fra la chiave specificata in ingresso alla funzione e tutte le potenziali minacce. Come detto precedentemente, minacce[] non contiene tutte le minacce, o meglio, non solo quelle. In minacce[] sono presenti tutte sequenze di due mosse bianche in grado di mattare.

```
            side=0;xside=1;
            if(!makemove(minacce[i][1].m.b)){
              takeback();
              takeback();
              return TRUE;
            }
```

Trovata qualche sequenza la cui prima mossa combacia con la chiave richiesta, dobbiamo eseguire la minaccia associata. Se questa mossa non fosse possibile, potremmo dire che la difesa si sia dimostrata tale da evitare il matto. Così verrà annullata la difesa, quindi la chiave e la funzione ritorna TRUE per indicare la validità della difesa.

```
                else{
                     ics=search(1);
                     if(!ics){
                          takeback();
                          takeback();
                          takeback();
                          return TRUE;
                     }
```

Tuttavia se la minaccia può ugualmente essere eseguita, viene chiamata la search() per stabilire se vi è scacco matto. Se il valore di ritorno è FALSE, ossia se non vi è matto, allora anche in questo caso la difesa si è dimostrata efficace e viene ritornato valore TRUE. A differenza del caso precedente viene chiamata per tre volte la funzione takeback() poichè bisogna annullare anche la minaccia appena eseguita.

```
                     else{
                          takeback();
                          continue; //magari blocca altre minac
                                    //ce per quella chiave
                     }
```

Se non viene rilevato alcuno scacco, non è ancora detto che vi possano essere altre minacce che questa difesa è in grado di sventare. Perciò verrà annullata solo la minaccia eseguita e si proseguirà con il ciclo. Una piccola modifica in questa sezione di codice permetterebbe il riconoscimento delle difese in grado di sventare tutte le minacce. Personalmente ritengo sufficiente sventare una sola minaccia per fare della mossa nera una difesa interessante.

```
                }
           }
     }
     takeback();
     takeback();
     // se esce dal ciclo, significa che non vi sono state

     //minacce sventate
     return FALSE;
}
```

Se non si esce dal ciclo atrtaverso una "return" allora nessuna minaccia è stata sventata da questa difesa. La funzione ritorna FALSE e la variante non dovrà essere visualizzata.

Naturalmente non tutti i problemi sono a minaccia, quindi non è utile fare questa valutazione in ogni problema, o comunque prima di stampare a video ogni output. Nei problemi a blocco, non vedremmo alcuna soluzione poiché non esistono minacce! In tal caso verranno mostrate all'utente

tutte le soluzioni ottenute senza compiere ulteriori controlli.
Per completezza ho pensato di aggiungere al menù proposto all'utente anche la possibilità di visionare ugualmente tutte le varianti, anche quelle che contengono difese che possiamo definire non efficaci. Quest'operazione non comporta alcun calcolo aggiuntivo, anzi evita i controlli richiesti per la rimozione delle difese non efficaci.

2.10 Identificatore di pattern

Un altro aspetto interessante della problemistica consiste nell'inserire all'interno del problema degli elementi tematici già riconosciuti e classificati dalla FIDE. Gli elementi tematici sono gli elementi combinativi generati sia dalla chiave che dalle difese o dalle varianti, essi stabiliscono le relazioni (statiche e di movimento) tra i vari pezzi della scacchiera e diventano quindi uno strumento fondamentale del cui lavoro usufruisce il compositore. E' proprio da qui che nasce l'esigenza di controllare che nei problemi creati siano presenti dei temi predefiniti e che a fronte di modifiche dei pezzi sulla scacchiera, tali temi rimangano preservati. Un supporto alla verifica dell'esistenza di un tema, potrebbe distogliere il compositore da questo oneroso compito spingendolo ad esprimere la propria iniziativa cercando nuovi elementi da aggiungere per rendere il problema più complesso.
La FIDE ha classificato numerosi elementi tematici e temi, ma riconoscerli tutti è un'impresa quasi impossibile oltre che di notevole difficoltà. Così ho pensato di estendere il programma permettendo l'individuazione di alcuni elementi tematici semplici, come il matto a stella, il matto a croce e la rosa di cavallo nero.

2.10.1 Il matto a stella

Nel gergo dei compositori, l'insieme delle case raggiungibili dal re prendono il nome di flights.
Il matto a stella o Starflights è un tema semplice che riconduce il suo nome alla forma disegnata delle possibili vie di fuga del re. In questo genere di tema, il re nero ha solo 4 modi per evitare il matto e sono esattamente le 4 mosse diagonali. Se si potesse unire con delle linee immaginarie la posizione del re con le sue flights, otterremo proprio una stella o meglio, una “star”.

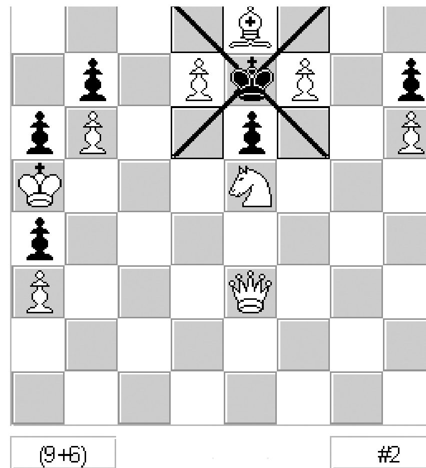


Figura 16: StarFlights

Si noti innanzitutto la simmetria presente in questo problema, soprattutto nella zona della scacchiera in cui risiede il re nero. I movimenti dei pedoni e dell'alfiere assicurano che il re nero non possa aprirsi vie di fuga catturando uno dei pezzi bianchi che lo circondano. Il pedone nero in e6 autoblocca il proprio re in quanto non può avanzare poiché fermato dalla presenza del cavallo in e5.
Dopo la chiave De4, il nero deve necessariamente muovere il proprio re poiché non esiste altro pezzo in grado di essere spostato. Trattandosi di un problema a blocco, il bianco non sarebbe in grado di mattare senza la sua mossa. Se il nero scegliesse una delle due vie di fuga in ultima traversa, il bianco matterebbe spostando la regina nell'unica casa in cui può minacciare il re nero. I pedoni in sesta e settima traversa bloccherebbero ogni via di fuga.
Se invece il re nero scegliesse una delle due flights in sesta traversa, uno dei due pedoni potrebbe raggiungere l'ottava traversa e una sua promozione ad alfiere causerebbe lo scacco matto.

```
C:\TESI\TESI.exe
benvenuti!

PROGRAMMA A SUPPORTO DEI PROGETTISTI DI PROBLEMI SCACCHISTICI
INSERISCI IL NOME DEL FILE:starflights.fen
APRO IL FILE: starflights.fen

8  . . . . B . . .
7  . p . P k P . p
6  p P . . p . . P
5  K . . . N . . .
4  p . . . . . . .
3  P . . . Q . . .
2  . . . . . . . .
1  . . . . . . . .

   a b c d e f g h

'on' per avviare
'd' per mostrare la scacchiera
'new' per cambiare problema
'bye' per uscire
>on
-------------------------------------------------
ECCO TUTTE LE SOLUZIONI:
SOLUZIONE 1:e3e4

CHIAVE: e3e4 [BLOCCO]
        VARIANTI:
        e7d8    -->     e4h4#
        e7f8    -->     e4b4#
        e7d6    -->     f7f8b#
        e7f6    -->     d7d8b#
MATTO A STELLA
-------------------------------------------------
Esaminati 1813 nodi in 4.700000 millisecondi

'on' per avviare
'd' per mostrare la scacchiera
'new' per cambiare problema
'bye' per uscire
>
```

Figura 17: Output del problema Starflights

Una volta stampate tutte le chiavi e tutte le rispettive varianti, verrà visualizzato un messaggio che indica l'elemento tematico riconosciuto.

2.10.1.1 Il codice

Riconoscere elementi tematici di questo tipo è piuttosto banale se si ha a disposizione gli strumenti adatti. Durante il calcolo dello scacco matto #2, ho pensato di non limitarmi a presentare solo le soluzioni di trovate lungo la scansione dell'albero, ma di salvarle in strutture dati in modo da poter compiere operazioni di ricerca e confronto in modo semplice e veloce. Ebbene, riconoscere elementi tematici come il matto a stella e i due successivi è un'operazione che richiede semplicemente l'analisi del contenuto di un vettore.
La seguente istruzione di controllo è inserita all'interno di un ciclo che legge tutte le difese del nero. Non appena si incontra una difesa che coinvolge il re, viene chiamata la funzione mattostella() che valuta se la flights può essere una delle quattro necessarie per il riconoscimento del tema.

```
    if(piece[matti[u][1].m.b.from]==5){//se il pezzo di questa
                                       //difesa è un re
    mattostella(matti[u][1].m.b.from,matti[u][1].m.b.to);
```

```
void mattostella(int partenza, int destinazione){
    int diff;
```

Mattostella() è una funzione che riceve in ingresso la casa di partenza e quelle di destinazione del re.

```
    diff=partenza-destinazione;
```

Effettuare la differenza tra la casa di partenza e la casa di destinazione del re, permette di capire quale delle 8 possibili vie di fuga del re è stata scelta. Infatti, la scacchiera è stata definita non come una matrice 8x8 ma come un vettore di 64 elementi.

```
    switch(diff){
        case 9:{            //in alto a sx
            d1=TRUE;
            break;
        }
        case 7:{            //in alto a dx
            d2=TRUE;
            break;
        }
        case -7:{           //in basso a sx
            d3=TRUE;
            break;
        }
        case -9:{
            d4=TRUE;
            break;
        }
        default:{
            altri=TRUE;
            break;}
    }
}
```

Nel caso del re ci possiamo attendere quindi 8 possibili valori della variabile "diff"(le 8 flights), di cui solo quattro possono formare la stella e sono i primi citati nelle scelte alternative. Se una di queste situazioni si verifica, una specifica variabile globale viene settata a TRUE.
Trovare una differenza che non coincide con i 4 valori indicati, significa che il re ha trovato una via di fuga alternativa che non permetterebbe la rappresentazione di una stella. Questo caso fa cadere tutti i casi precendenti, poichè un matto a stella deve avere solo 4 vie di fuga in posizioni specifiche.

Per poter stampare a video il messaggio, è necessario verificare la seguente condizione in relazione ad ogni chiave:

```
if(d1&&d2&&d3&&d4&&(!altri))
    printf("MATTO A STELLA\n");
```

Il codice appena proposto segue lo stesso schema anche nei due temi successivi. Per questo motivo mi riservo di riportare il codice delle funzioni per la verifica del matto a croce e del problema a rosa di cavallo nero.

2.10.2 Il matto a croce

Molto simile al precedente, il matto a croce permette le sole vie di fuga orizzontali e verticali in modo da formare una croce. Poiché il matto a croce non discosta di molto dal matto a stella, se non per l'orientamento delle vie di fuga, non verrà proposto un esempio. In molte situazioni, tale matto può essere individuato col nome di “Plus mate”.
L'implementazione del riconoscimento del matto a croce segue lo stesso meccanismo indicato per il matto a stella.

2.10.3 Rosa di cavallo nero

Una rosa di cavallo nero (o anche Knight-Wheel) è un tema in cui le difese nere contengono tutte le 8 possibili mosse del cavallo.

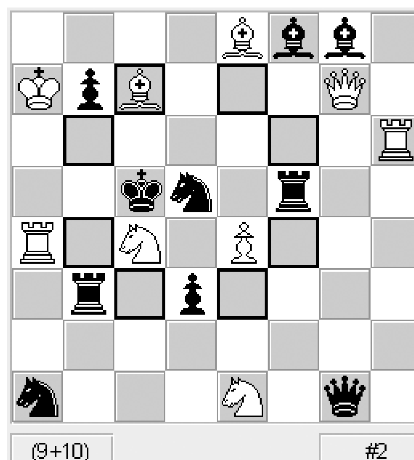


Figura 18: Knightwheel

```
C:\TESI\TESI.exe
benvenuti!

PROGRAMMA A SUPPORTO DEI PROGETTISTI DI PROBLEMI SCACCHISTICI
INSERISCI IL NOME DEL FILE:knightwheel.fen
APRO IL FILE: knightwheel.fen

8   . . . . B b b .
7   K p B . . . Q .
6   . . . . . . . R
5   . . k n . r . .
4   R . N . P . . .
3   . r . p . . . .
2   . . . . . . . .
1   n . . . N . q .

    a b c d e f g h

'on' per avviare
'd' per mostrare la scacchiera
'new' per cambiare problema
'bye' per uscire
>on
-------------------------------------------------
ECCO TUTTE LE SOLUZIONI:
SOLUZIONE 1:c4a5

CHIAVE: c4a5 [a4c4#]
        VARIANTI:
        b7b5    -->     h6c6#
        d5c7    -->     g7c7#
        d5e7    -->     c7d6#
        d5b6    -->     a5b7#
        d5f6    -->     g7f8#
        d5b4    -->     c7b6#
        d5f4    -->     g7g1#
        d5c3    -->     e1d3#
        d5e3    -->     g7d4#
        b3b4    -->     e1d3#
        b3c3    -->     a5b7#
        g1d4    -->     g7d4#
BLACK KNIGHT-WHEEL
-------------------------------------------------
Esaminati 10341 nodi in 40.600000 millisecondi
```

Figura 19: Output problema Knightwheel

Il problema presentato è a minaccia ed ha una sola chiave: Ca5. Fra le possibili difese troviamo tutte le mosse del cavallo nero, ma non è detto che debbano esistere solo quelle! In questo caso esistono quattro altre difese oltre all'intera ruota del cavallo nero. Tutte queste difese sono efficaci, nel senso che riescono a sventare una o più minacce (in questo l'unica).
Il programma è in grado di rilevare questa situazione, in quanto opera un confronto fra le difese nere ed è in grado di stabilire se fra quelle ne esistono 8 che coinvolgono il cavallo.
Dopo aver presentato tutte le chiavi e tutte le varianti, verrà lanciato un messaggio che indica il pattern riconosciuto.

2.11 CENNO AGLI OUTPUT

TSCP non gode di interfaccia grafica propria perciò stampa a video la scacchiera e le mosse che i due giocatori effettuano tramite interfaccia di carattere. Allo stesso modo ho presentato gli output delle elaborazioni dell'intero programma.
Per migliorare la leggibilità, ma anche l'usabilità, ho dato all'output una formattazione che cerchi di andare incontro alle esigenze dei compositori.
Prima di tutto, il software espone tutte le chiavi del problema e se queste sono più di una, verrà anche segnalato un messaggio di “Problema demolito”.
Un avviso appare anche nel caso in cui il problema non abbia alcuna soluzione.
Il passo successivo consiste nel presentare le informazioni relative ad ogni chiave, quindi, per ogni soluzione trovata, dovranno apparire: la chiave, le minacce, le difese ed i matti. Sono i quattro elementi che compongono il problema di scacco matto #2.
Al compositore è gradito vedere a fianco della chiave, la rispettiva o le rispettive minacce. Per non confondere la scrittura con altri elementi di valore, quest'ultime verranno scritte fra parentesi quadre. Se il problema è a blocco, apparirà una notifica al posto delle minacce.
Come ormai constatato, ogni chiave apre diverse difese, e quindi varianti. Sarebbe opportuno scriverle in corrispondenza della chiave a cui fanno riferimento. Le varianti ottenute vengono così elencate una sotto l'altra immediatamente dopo la chiave.
Ultime informazioni visualizzate sono dati statistici riferiti all'elaborazione compiuta, come il numero di nodi visitati durante la ricerca ed il tempo totale richiesto per portare a termine il carico di lavoro.

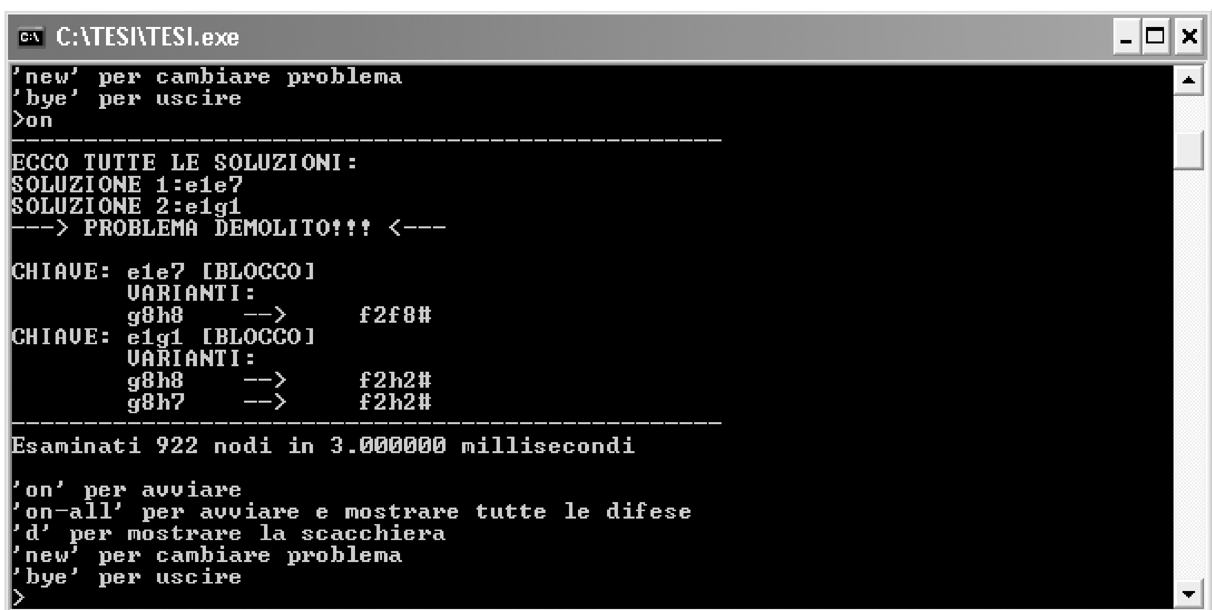


Figura 20: Un possibile output

conclusione

1. RISULTATI OTTENUTI

L'algoritmo per il calcolo dello scacco matto diretto in 2 mosse progettato come oggetto della tesi si è dimostrato funzionale in quanto ne sono stati confrontati gli output con quelli di un software analogo e già diffuso nel mondo dei compositori di problemi scacchistici. Sto parlando di Problemist: un software free disponibile su internet per piattaforma Windows [6]. Inoltre, molti dei problemi sottoposti a verifica provengono dal libro scritto da John Rice, ovvero il presidente della PCCC[4], l'organo della FIDE che si occupa di gestire le competizioni fra compositori. Oltre alle soluzioni dei problemi proposti, sono presenti anche molte descrizioni che mi hanno aiutato nel controllo delle funzionalità del programma.
Alla luce di tutto questo non sono in grado di dimostrare la completa validità del programma: un'esaustiva fase di test richiederebbe molto tempo e non essendo un problemista, farei molta fatica a trovare eccezioni o casi particolari con cui mettere alla prova l'algoritmo. Tuttavia sono state effettuate numerose verifiche con problemi di vario genere: dai problemi a minaccia a quelli a blocco, da quelli a chiave unica a quelli con più chiavi, quelli con matto in una mossa, quelli con un tema specifico. In qualunque situazione il programma ha generato un'elaborazione corretta che ha portato a risultati soddisfacenti in grado di adempiere le richieste dell'utilizzatore in tempi decisamente brevi. Le elaborazioni prese in considerazione non superavano i 50 millisecondi su entrambe le macchine a mia disposizione. Ovviamente il tempo richiesto è direttamente proporzionale al numero di nodi effettivamente visitati e dipende anche dal tipo di macchina su cui l'applicativo viene lanciato.
Ritengo che il programma sia un buon strumento di analisi di problemi scacchistici, ma che tuttavia non possiede ancora elementi che ne agevolino la composizione.
La combinazione di composizione e analisi, farebbero dell'applicativo un potente tool a supporto dei compositori di problemi scacchistici.

2. ALTRI ELEMENTI DI ANALISI

Il software oggetto di questa tesi, ricopre una grande fetta del supporto di cui un compositore può aver bisogno, poiché risponde all'esigenza primaria di presentare tutte le soluzioni del problema corredate delle rispettive varianti. Qualora il compositore decida di intraprendere un'analisi ben più approfondita, si rende necessario aggiungere nuovi elementi in grado di ampliare il contenuto informativo associato ad un problema. Tali elementi consentirebbero al compositore di apportare nuove modifiche o di compiacersi del problema creato dal momento che non solo il programma sa trovare soluzioni, ma è in grado di estrarre componenti nascoste e altrettanto difficili da individuare.
Sarebbe quindi opportuno fare un altro passo verso il compositore arricchendo l'applicativo con delle sezioni indispensabili per un'analisi dettagliata.

Tra i fattori di spicco che potrebbero migliorare la qualità del software emergono:

- Il gioco virtuale: il gioco che si sviluppa in seguito di una mossa bianca

• Il gioco apparente: il gioco che si sviluppa immaginando che il bianco abbia già giocato la chiave

Le definizioni di questi due elementi scostano leggermente dal quello che è il calcolo della soluzione di un problema di scacco matto diretto in 2 mosse. Tuttavia, poter leggere il gioco a fronte di questi termini presenterebbe una vantaggiosa panoramica dei lineamenti che definiscono il problema.

Come già chiarito nel capitolo precedente, il programma sviluppato è in grado di riconoscere alcuni elementi tematici semplici. Eventuali modifiche al problema durante la progettazione, potrebbero causare la perdita degli elementi tematici che lo caratterizzano. E' proprio per questo motivo che si rende utile la presenza di "riconoscitore di temi". La validità di questa feature è tanto maggiore quanti sono i temi o gli elementi tematici che si è in grado di distinguere.
Per quanto possa sembrare banale riconoscere "matti a stella" o "kinghtwheel" (il codice presentato ne sono una dimostrazione), trovare combinazioni di elementi tematici che compongono un tema è ben più difficile! A differenza di quanto visto per gli elementi tematici semplici, riconoscere temi richiede una valutazione delle mosse di un particolare pezzo, ma il tutto deve essere valutato in relazione con gli altri pezzi amici e nemici. Si pensi al tema Grimshaw. Tale tema è caratterizzato dalla dall'interferenza reciproca che si crea tra una torre ed un alfiere. Quindi non è necessario prendere in considerazione la sola mossa di uno dei due pezzi, ma bisogna badare anche alle conseguenze che si ripercuotono sulle altre parti in gioco. Questo è solo un esempio ma credo che sia sufficientemente significativo.

3. ELEMENTI UTILI ALLA COMPOSIZIONE

Il programma è stato scritto con l'intento di sviluppare un algoritmo in grado di calcolare tutte le soluzioni di un problema. Per questo motivo, l'applicativo è sprovvisto di una sezione in cui è possibile determinare le posizioni dei pezzi parte della composizione dell'utente. Benché i problemi in ingresso siano scritti in notazione standard, appare chiaro che il compositore debba trascrivere o generare il file di input prima di analizzarlo.
Risulta piuttosto scomodo, inoltre, dover studiare problemi senza godere di un'interfaccia grafica che permetta una visualizzazione complessiva ed immediata. Il programma sviluppato presenta i pezzi sulla scacchiera attraverso lettere e l'interpretazione del loro significato è tutt'altro che spontanea. Se il programma godesse di un'interfaccia grafica in grado di rappresentare la scacchiera, si potrebbe creare un editor di problemi che consenta di posizionare o rimuovere pezzi ad arbitrio del compositore e quindi evitare le fasi di traduzione del problema in notazione FEN e di caricamento del problema da file su disco.
L'uso di un'interfaccia grafica introdurrebbe la possibilità di includere numerose altre opzioni a supporto della creazione di problemi.

4. ALTRE UTILITA'

L'attività del compositore spesso poggia su basi solide per poter dare origine a problemi ben più sofisticati. Un progettista frequentemente riprende problemi già esistenti ed applica modifiche

introducendo nuovi elementi che cerchino di rendere la composizione ancor più interessante. Per quanto un problema possa essere conosciuto, è ben poco probabile che il compositore si ricordi l'ubicazione di tutti i pezzi che vi prendono parte. Si rende così necessaria l'esigenza di una sorta di database, una raccolta dei problemi più conosciuti e di quelli creati dal compositore stesso. Potrebbe essere di maggior utilità anche una classificazione di questi secondo parametri a discrezione dell'utilizzatore del programma. Dotare il software di funzionalità di gestione di database permetterebbe oltre ad una archiviazione intelligente, anche una rapida ricerca di problemi con determinate caratteristiche da cui prendere spunti per la composizione.

5. SVILUPPI FUTURI

Lo studio di problemi scacchistici, è un'attività che richiede molto tempo e concentrazione. Per questo motivo, i compositori vi si dedicano nel loro tempo libero in cui non vi è nessun altro genere di attenzione. Si pensi ad esempio a quando si deve affrontare un lungo viaggio in treno, piuttosto che un pomeriggio disteso sotto il sole di una località marittima. Il compositore potrebbe impiegare questo tempo studiando nuovi problemi anche se in queste situazioni non è sempre possibile avere a disposizione un supporto per il calcolo, ad esempio un PC, che permetta di agevolare l'attività costruttiva. Servirebbe infatti qualcosa di comodo, tascabile e non vincolato dalla sua collocazione nello spazio. Servirebbe qualcosa che possa essere sempre a portata di mano, facile da usare, che sia in grado di soddisfare tutti i requisiti in tempi ragionevoli e che permetta un utilizzo intuitivo anche a fronte delle ridotte dimensioni.
La recente tecnologia ha creato i "palmari" ossia delle macchine molto simili ai comuni computer, ma che, come dice il nome stesso, possono essere ospitati sul palmo di una mano. Macchine di questo genere colmano il bisogno di chi ha la necessità di portare con se uno strumento dalle elevate capacità elaborative indipendentemente dal luogo in cui si trova. I palmari vengono generalmente utilizzati come agende elettroniche, fogli di testo o di calcolo, internet e quindi una serie di applicativi di svariati generi, fra cui software per il gioco degli scacchi. Poter dare al compositore la possibilità di avere il programma anche su un palmare, gli permetterebbe di soddisfare la sua passione per gli scacchi in qualsiasi luogo o momento della giornata, senza il vincolo dei cavi di alimentazione o di ingombro dovuto alle dimensioni di un PC.

Ora che l'algoritmo è terminato, è mia intenzione occuparmi innanzitutto di aggiungere tutti gli elementi di analisi e composizione descritti in questo capitolo e che per motivi di tempo non sono riuscito ad inserire durante lo sviluppo del programma. A lavoro ultimato vorrei tentare il porting dell'applicazione verso macchine come i palmari. La "distanza" tecnologica fra ambiente di origine e ambiente di destinazione è significativa e l'operazione di porting potrebbe essere tutt'altro che banale ma sicuramente un'esperienza utile per la mia crescita in ambito informatico.

RINGRAZIAMENTI

Vorrei ringraziare tutti coloro che mi hanno seguito durante questi sei mesi di lavoro ed in particolare il mio correlatore Marco Guida che si è sempre dimostrato disponibile per qualsiasi chiarimento.
Un ringraziamento va anche ai docenti che mi hanno saputo orientare nella stesura del software e nell'utilizzo dell'ambiente di sviluppo impiegato.

BIBLIOGRAFIA
John Rice , “Chess Wizardry, The New ABC of Chess Problems”, B.T. Bastford Ltd, London

SITOGRAFIA

SITI PRINCIPALI
[1]Enciclopedia on-line
http://it.wikipedia.org

[2]Appunti sulla problemistica
http://xoomer.virgilio.it/livornoscacchi/problemistica.htm

[3]Sito dell'ente supremo che si occupa di competizioni scacchistiche continentali
http://www.fide.com

[4]Sito ufficiale della Commissione Permanente della FIDE per la Composizione Scacchistica
http://www.saunalahti.fi/~stniekat/pccc/index.htm

[5]Il software TSCP
http://home.comcast.net/~tckerrigan/

[6]Il software Poblemiste
http://perso.wanadoo.fr/problemiste/

Rivista “on-line” di problemi di scacchi
http://christian.poisson.free.fr/problemesis/numero48.html

ALTRI SITI
Alcuni termini scacchistici
http://scacchi.qnet.it/

Appunti su algoritmi per il gioco degli scacchi
http://www.itportal.it/developer/algoritmi/scacchi/

Scacchi e macchine
http://www.mondodigitale.net/Rivista/05_numero_quattro/Ciancarini_p._3-16.pdf

Dettagli sulla chess programming
http://www.valocchi.it/lamosca/

Sito per scacchisti programmatori
http://www.gsei.org

http://www.chessolympiad-torino2006.org/comunicazione_gg_millionaire.php