

2011

Algorismes aplicats als escacs: Programació d'una aplicació



INS MANOLO HUGUÉ

04/02/2011

A tot aquell que hagi fet el treball possible:

for(int i = 0; i ≥ 0; ++i) gràcies

TREBALL DE RECERCA

2º BATXILLERAT 'A'

4 DE FEBRER DE 2010

INS MANOLO HUGUÉ

DEPARTAMENT DE TECNOLOGIES



ÍNDEX

1. Introducció	3
1.1. Justificació	3
2. Intel·ligència artificial i jocs	4
2.1. Tipus de jocs.....	4
2.2. Breu evolució de la màquina en els escacs.....	5
3. Representació del taulell i dels moviments	8
3.1. Llista de peces	8
3.2. Manera simple: array de 8x8	8
3.3. Bitboards	10
3.4. Comparativa: avantatges i inconvenients	12
3.5. Què és un moviment?.....	12
4. Algoritmes.....	14
4.1. Generació de moviments.....	14
4.2. Minimax: característiques.....	18
4.3. Millores sobre Minimax.....	21
4.3.1. Poda Alpha beta.....	21
4.3.2. Negamax.....	25
4.3.3. Fer petit l'interval de cerca	26
4.3.4. Ordenar les jugades	26
4.3.5. Profundització iterativa	28
4.3.6. Evitar l'efecte horitzó: la cerca de la quietud.....	29
4.4. Avaluació estàtica de la posició.....	30
4.5. Taules de transposició: el <i>hash</i>	33
5. Altres millores	36
5.1. Llibre d'obertures.....	36
5.2. Taules de finals.....	36
5.3. Autoregulació del temps.....	37
5.4. Log de la partida, el PGN.....	38



ALGORISMES APLICATS ALS ESCACS: PROGRAMACIÓ D'UNA APLICACIÓ

5.5. Protocols UCI i Xboard	39
5.6. Oferiments de taules i rendició	40
6. Entrevista amb Robert Hyatt	42
7. Conclusions	45
8. Bibliografia	46
9. Annexes	48
A. Entrevista amb Robert Hyatt.....	48
B. Contingut del CD	52



1. INTRODUCCIÓ

1.1. JUSTIFICACIÓ

Aquest treball de recerca té com a objectiu la implementació d'una aplicació que jugui a escacs. Al llarg del treball s'explicarà de manera genèrica, sense entrar en detalls concrets, com s'han fet els principals procediments que permeten que aquesta aplicació jugui a escacs.

Aquest tema el vaig proposar com a iniciativa pròpia, ja que cap altre tema em satisfia, i el professor d'electrotècnia, Josep Ferràndiz, el va acceptar, tot i que oficialment no l'ha pogut tutoritzar, ha estat ell qui ha fet el seguiment de l'evolució del treball. Gràcies a l'ajuda de la meva tutora final, que va ser qui finalment va acceptar el treball, ha estat possible dur-lo a terme. El tema el vaig escollir perquè des de fa anys sóc jugador d'escacs federat i dins d'aquest món sempre hi ha l'argument "aquesta jugada me la va dir la màquina, per tant, és la jugada bona", i mai em va acabar de convèncer aquest argument perquè no n'era conscient de tota la "feina" que fan aquests jugadors de silici. Aquest treball m'ha servit, doncs, per a investigar en l'"intel·ligència artificial" i m'ha convençut que aquestes màquines calculen variants d'escacs de manera molt precisa, millor que qualsevol home.

El treball està dividit en tres grans apartats: el primer és fer l'abstracció del joc perquè una màquina el pugui entendre que és cada cosa, el segon és el desenvolupament de l'algoritme Minimax i el tercer es proposen diverses millores sobre aquest algoritme per a fer-lo totalment rendible.

He hagut de fer dues recerques d'informació per a dur a terme aquest programa. La primera ha estat per a aprendre a programar en un llenguatge qualsevol, en aquest cas va ser en C++ ja que dins de la Olimpíada Informàtica Espanyola es va fer un curs i aquest va ser el llenguatge explicat. Amb aquest curs i fent molts problemes a la seva web (més de 90) vaig ser capaç de dominar la part més bàsica de la programació i algorísmia en C++. Un cop vaig ser capaç de fer petits programes intel·ligents com el connect4 i el tic-tac-toe (3x3) (adjunts al Annex), vaig començar a fer el meu propi programa que juga a escacs. Després, va començar la segona recerca, aquest cop buscant algorismes d'altres programadors per a fer motors d'anàlisi.



2. INTEL·LIGÈNCIA ARTIFICIAL I JOCS

Primer de tot m'agradaria contextualitzar els escacs dins del món dels jocs, i els avenços que han experimentat des del seu inici dins el món de les màquines.

2.1. TIPUS DE JOCS

Els jocs, dins la intel·ligència artificial, no són més que un "problema", ja que contenen les característiques bàsiques d'aquests:

- Un inici, on hi ha elements que defineixen el problema.
- Accions que es poden fer per a resoldre el problema.
- Un punt final, què és el que es vol aconseguir?

Però quina és la peculiaritat que tenen els jocs perquè els puguem considerar una categoria diferent? Doncs que en els jocs direm que són problemes que es plantegen com una "cursa" per a aconseguir el punt final entre dos o més jugadors, o agents.

Fins aquí, és fàcil veure aquestes característiques dins dels escacs: l'inici és la posició i les normes al començar la partida, les accions per a "resoldre" són els moviments i l'estat final és l'escac i mat que té definició pròpia.

Veient això també es poden fer subdivisions dins d'aquests tipus de problemes anomenats "jocs". És important saber que ens centrarem dins els jocs d'informació perfecta, on tots els jugadors saben ho tot i per tant, no hi ha atzar (a diferència del pòquer). Per tant, inclourem els escacs dins d'aquests tipus de jocs, jocs deterministes.

Una altra cosa que particularitza els escacs és el nombre de jugadors i la relació que hi ha entre ells: el fet que siguin dos que volen aconseguir el mateix objectiu fa que les coses que són bones per a un jugador siguin dolentes per al rival. Si sumem els avantatges d'un jugador més els de l'altre ens haurà de donar zero perquè no pots aconseguir un avantatge sense perjudicar en el mateix nivell al teu adversari. D'aquests tipus de jocs s'anomenen de suma zero.

Així, des d'un punt de vista formal, direm que els escacs s'inclouen dins dels jocs bipersonals amb informació perfecta i de suma zero, com altres jocs que podrien ser les dames, l'Othello, el Go, el quatre en ratlla, el Go-Moku o fins i tot el senzill tres en ratlla, entre d'altres. Això ens és útil perquè per a resoldre qualsevol d'aquests jocs podem utilitzar els mateixos algorismes, amb les mateixes idees inherents.

Per tal de distingir la complexitat de cada joc, des d'un punt de vista computacional, només s'ha de mirar el nombre de posicions, o estats intermedis, que hi ha entre l'inici i el final. Així, el tres en ratlla té una complexitat molt baixa, ja que poden existir 26.830 partides diferents,



ALGORISMES APLICATS ALS ESCACS: PROGRAMACIÓ D'UNA APLICACIÓ

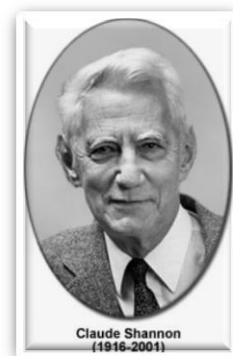
en canvi en els escacs, la complexitat augmenta exponencialment segons el nombre de jugades, i si fem la aproximació a 40 moviments, serà de 10^{120} partides diferents.

2.2. BREU EVOLUCIÓ DE LA MÀQUINA EN ELS ESCACS

Ja des del 1700 els inventors van somiar amb la construcció d'una màquina que jugués a escacs encara que no es va aconseguir. A títol de curiositat, Von Kempelen va construir una màquina anomenada "El Turc" que amagava a un jugador dins seu i simulava jugar. Aquest frau va existir fins a 1850 quan es descobrí la trampa. Tot i així aquest "jugador" va aconseguir guanyar Napoleó en jocs de guerra.

El primer enginy que aconseguí jugar a escacs l'inventà Torres Quevedo al 1912, i per mitjà d'electroimants movia les fitxes i tenia un algoritme fet a partir d'elements electromecànics. Tot i així "El ajedrecista", que era el nom de l'autòmat, només era capaç de jugar el final de torre i rei contra rei. Cal afegir que guanyava sempre, però no de la manera més òptima.

Al 1950 els treballs de Claude Shannon i d'Alan Turing foren publicats. Shannon posà les bases teòriques dels programes d'escacs, on hi apareixen idees innovadores com la funció d'avaluació i de la cerca de la quietud. Fins i tot va fer una divisió entre les futures màquines: les màquines de tipus-A que són força bruta, que es basaven en calcular-ho tot, i les de tipus-B on el mateix programa s'autoregula per decidir com de profund ha d'investigar. Shannon també va determinar que les màquines tipus-A no serien capaces de resoldre els escacs perquè va predir el número de variants que haurien de calcular: 10^{120} , suposant que les partides duressin 40 jugades i cada jugador pogués escollir 30 moviments a cada posició. Aquest número 10^{120} , és extret de fer $(30 \times 30)^{40}$, és a dir, $900^{40} = 10^x$, és a dir, $x = 40 \cdot \log 900 \approx 120$, per aquest fet, a 10^{120} s'anomena "nombre de Shannon".



Claude Shannon, Alan Turing i John McCarthy a més de dissenyar algoritmes que juguessin a escacs, van fer aportacions al món de la intel·ligència artificial: Shannon és el pare de "La teoria de la informació", Turing dissenyar un test per a determinar si un sistema podia ser intel·ligent i McCarthy va utilitzar el terme "intel·ligència artificial" per primer cop i ha dissenyat el llenguatge de programació "Lisp".



ALGORISMES APLICATS ALS ESCACS: PROGRAMACIÓ D'UNA APLICACIÓ

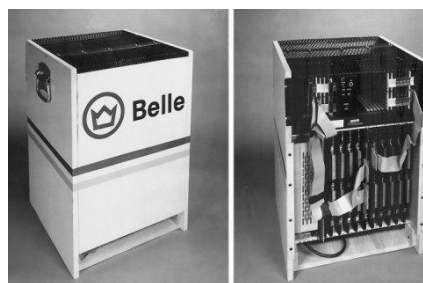
Turning per la seva part, a la universitat de Manchester, va construir el primer programa que jugava a escacs, tot i que només arribava a pensar a una jugada de profunditat. A més, tant Shannon com Turning, van treballar amb l'algoritme minimax i una funció d'avaluació, cosa que influencià tot el futur. Aquests dos investigadors van intentar crear motors d'anàlisi i van fer aportacions teòriques que formen les bases per als programes de les properes generacions.

A partir de la segona meitat del segle XX, molts investigadors van començar a dissenyar els seus propis analistes. Tot i que van haver molts avenços, els més coneguts i significatius van sorgir dels programes ideats pels nord-americans Alan Kotok i John McCarthy i el programa desenvolupat a l'Institut de Física Experimental i Teòrica de Moscou (ITEP) que el dissenyaren grans jugadors. Aquests dos programes s'enfrontaren al 1961. John McCarthy va ser el primer en introduir la poda Alpha-Beta (" $\alpha\beta$ pruning": va permetre no examinar totes les jugades). Tot i que el treball de McCarthy era més complet des d'un punt de vista teòric, el match es va jugar a 4 partides amb 3 nivells de profunditat per al programa de McCarthy/Kotok i 4 nivells de profunditat del programa de l'ITEP, la victòria se la van emportar els russos per 3-1 (2+ 2= 0-).

Un cops entrats a la dècada dels 70 hi va haver molts avenços: els programes aconseguiren una profunditat de 7 mitges jugades, es van desenvolupar torneigs per a màquines, algunes ja implementaven taules de transposició... Fins i tot un Mestre Internacional dels escacs, David Levy, va jugar un match a 6 partides contra CHESS 4.7 on la màquina va aconseguir posicions guanyadores però no va saber rematar. El resultat de Levy-CHESS 4.7 va ser 3.5-1.5 (3+ 1= 1-). Aquell match causà la sensació que en breu les màquines que jugaven a escacs estarien per sobre dels millors jugadors del món. Cal destacar que als anys 70 va aparèixer una nova forma de pensar els escacs i orientar-los, eren els bitboards. Això va fer augmentar considerablement la velocitat dels ordinadors. Més endavant es descriu amb detall en què consisteix la tècnica dels bitboards.

Un altre avenç important va ser la creació de BELLE al 1973 que posseïa hardware dedicat a tasques específiques amb l'objectiu d'accelerar els càlculs. Va obtenir gran quantitat de premis als principis dels 80. Això va fer que les següents màquines dedicades als escacs se centressin en l'avenç del hardware, com ho va demostrar la màquina que el guanyaria, CrayBlitz.

A més, a la dècada del 80 hi va haver una gran expansió dels escacs per ordinador, degut a la reducció de costos del maquinari. Aquest fet va permetre tenir més efectius per a realitzar proves. Això es concretà al 1988, quan "Deep Thought" va guanyar un torneig on hi participaven jugadors humans de nivell d'elit mundial, com el Gran Mestre Brent Larsen que



Belle va ser dissenyada per a jugar a escacs



ALGORISMES APLICATS ALS ESCACS: PROGRAMACIÓ D'UNA APLICACIÓ

sofrí la primera derrota dels "humans d'elit" contra la màquina. Un altre GM, Gary Kasparov va jugar al 1989 contra Deep Thought aconseguint dues victòries i cap derrota, deixant clar que l'home era superior a la màquina.

El principal desenvolupament a la del 90, degut al moviment mediàtic que va provocar, va ser l'enfrontament entre l'ordinador Deep Blue i el campió del món Gary Kasparov. L'empresa IBM va crear Deep Blue i va fer gran quantitat de propaganda. El primer match disputat al 1996 el guanyà Kasparov per 4 partides a 2 (3+ 2= 1-). L'evolució de Deep Blue va ser estratosfèrica i al



Deep Blue guanyà per primer cop a la història al campió del món, Garry Kasparov

1997 es va fer el rematch. La màquina va aconseguir guanyar a Kasparov per la mínima, 2.5-3.5 (1+ =3 -2). Desgraciadament el sistema de Deep Blue no era gens innovador, utilitzava la cerca de jugades de manera bruta perquè tenia un gran hardware que li permetia examinar dos-cents milions de posicions per segon i a més disposava de llibres d'obertures fets a mà i taules de finals molt precises. Per a fer una idea de la potencia d'aquesta màquina podem dir que era el 259è superordinador amb més potència de tot el món, amb 11,38 gigaflops. Tot el què DeepBlue feia ja s'havia estudiat abans i no va aportar res de nou, en canvi demostrà la força que podien arribar a tenir els ordinadors sense algorismes complexos.

Des d'aleshores fins a l'actualitat els programes han guanyat la majoria de matchs entre homes i màquines.

Actualment la investigació avança amb la idea de fer córrer aquests programes en màquines com les de casa, aconseguint que en un ordinador qualsevol puguem tenir el nivell de Deep Blue. Aquests avenços han estat possibles gràcies a que els programes han "descobert" com reduir el nombre de jugades a examinar fent l'arbre molt més petit. Per desgràcia, la majoria de programes actuals s'han dissenyat amb finalitats comercials i els seus codis són secrets. Aquests programes comercials s'enfronten en matchs entre ells i contra diversos GM's, aconseguint bons resultats, però tot amb la idea de vendre el seu producte al gran públic. El programa més fort i de codi lliure és Crafty, que és l'evolució de CrayBlitz, dissenyat per Robert Hyatt. Podeu consultar l'entrevista que vaig fer a Robert Hyatt als annexos.

A més, a dia d'avui els jugadors d'escacs de tots els nivells consideren els programes una ajuda per a verificar les anàlisis pròpies, no per lluitar contra ells, ja que han demostrat la seva superioritat envers l'home.



3. REPRESENTACIÓ DEL TAULELL I DELS MOVIMENTS

3.1. LLISTA DE PECES

Els primers programes ideats abans de Shannon i Turing, es basaven en una llista on tenien la ubicació de totes les peces, tant pròpies, com les del rival. Guardaven dues llistes, una amb les peces pròpies i un altre amb les rivals. S'havia arribat a l'acord que la posició 1 era per al rei, la 2 per a la dama... fins a completar totes les peces.

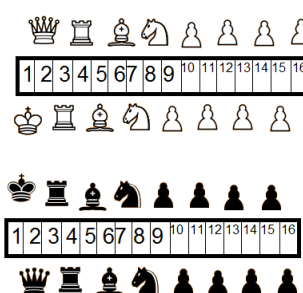
Això era degut a que la memòria de la computadora era molt limitada. El gran avantatge que rebien és que per saber on es trobava el rei dels jugadors el temps era instantani. Per desgràcia aquesta formació tenia moltes carències a l'hora de coronar i era contraproductiu, necessitava més memòria per dir en quina peça es transformaria i on col·locar-la dins la seva pròpia llista. Una altra dificultat afegida era que els números que es guarden en la llista són quadres numerats que estan de manera abstracta a la ment del programador, no dins la màquina. Tenia, a més, el defecte que li era més complicat guardar factors inherents a la posició i que no es derivava de les peces, com el torn, la possibilitat d'enroc o el compte de les 50 jugades per a fer taules. Per exemple, poden existir dues posicions idèntiques, però que en una d'elles el rei blanc no s'ha mogut abans i per tant es podrà enrocar, i un altre igual on el rei sí que s'ha mogut i no ho podrà fer.

Aquest sistema està en desús.

3.2. MANERA SIMPLE: ARRAY DE 8X8

Des d'un principi Claude Shannon va proposar la idea més senzilla, utilitzar una taula de 8x8 amb llocs extra a la memòria al final per a guardar altres factors intrínsecs de la posició, com els enrocs curts, llargs, tant de blanques com de negres, un bit per especificar el torn, el compte de 50 jugades per a fer taules i el quadre on el peó pot prendre al pas.

Cada quadre d'aquesta taula pot contenir uns valors enters entre -6 i +6 que simbolitzarà una peça. Així doncs si hi ha un peó blanc tindrà el valor 1, si hi ha un cavall blanc 2, alfil blanc, 3, torre blanca 4, dama 5 i rei blanc 6. Els valors negatius correspondran a les peces negres (peó negre -1, Cavall negre -2...). El valor '0' quedarà reservat per a caselles on no hi ha cap peça.





ALGORISMES APLICATS ALS ESCACS: PROGRAMACIÓ D'UNA APLICACIÓ

En el meu programa he utilitzat aquesta idea, on en comptes de tenir una matriu de 8x8 més els factors extres, he utilitzat una llista de 71 elements (que són "caràcters"), on ordenats del 0 al 70, estan totes les dades interessants per a definir que és una posició:

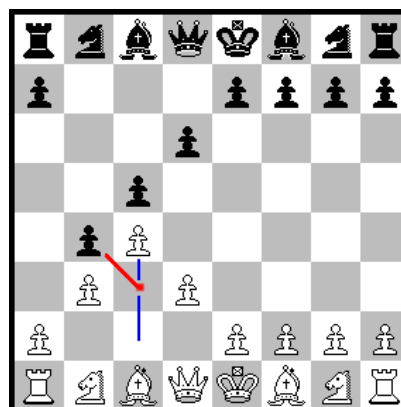
- Les posicions del 0 al 63 corresponen als quadres, aporten la informació codificada tal com s'ha dit abans (1=peó blanc, 2=Cavall blanc...). La posició 0 correspon al quadre h1, la posició 1 al quadre g1 (...), la posició 7 al quadre a1, la posició 8 al quadre h2 (...) fins al quadre a8 que correspon a la posició 63 de la meua llista. Saber a quina fila i columna es troba cada quadre es fa fàcilment amb les operacions de divisió entera (depreciant el residu, per molt alt que sigui) i amb el mòdul, és a dir, el residu de la divisió entera. D'aquesta manera es pot expressar així:

columna = quadre mod 8, fila = quadre/8, i així tenim la seva posició en una taula de 8x8, numerat de 0 a 7.

- Les posicions de la 64 a la 67 indiquen els drets d'enroc, i actuen com a dades booleans, certes o falses. La 64 indica si el blanc pot enrocar-se pel flanc de rei, la 65 pel flanc de dama i les 66, 67 de la mateixa forma indiquen el flanc de rei i de dama per a les negres.
- La posició 68 actua com un booleà, indica cert (true) o fals (false) a la pregunta "Li toca a les blanques?". D'aquesta manera la posició 68 ens informa de qui té el torn.
- Per a fer més fàcil la generació de moviments, la posició 69 ens diu si en l'última jugada s'ha mogut un peó dos cops cap endavant, i si això és cert (és a dir té un valor diferent de 0) ens dirà en quina posició del taulell és pot capturar al pas aquest peó. Per exemple a la figura del diagrama. l'última jugada de les blanques és c4, doncs el programa guardarà a la posició 69 del taulell la casella c3, per a què el peó negre pugui matar al pas.

4	2	3	6	5	3	2	4
1	1	1	1	1	1	1	1
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
-1	-1	-1	-1	-1	-1	-1	-1
-4	-2	-3	-6	-5	-3	-2	-4
X	X	X	X	X	X	X	X

Segons el conveni agafat aquesta seria la representació del taulell en una taula de 8x8, on les X són els llocs per a guardar informació extra, on cada programador escollirà l'ordre





ALGORISMES APLICATS ALS ESCACS: PROGRAMACIÓ D'UNA APLICACIÓ

- Per últim i per a satisfer una norma dels escacs: "Si passen 50 moviments (50 blancs i 50 negres) sense que cap jugador capturi una peça rival, o sense que un jugador avanci un peó, la partida finalitzarà en taules". La posició 70 guardarà el número de jugades que fa que no es captura cap peça i que no s'avança un peó, si aquest número és més gran o igual a 100 (50 moviments blancs i 50 negres) la partida acaba en taules.

Aquesta és la forma que té el programa que he dissenyat per a guardar el que és una "posició" dins del seu cervell.

3.3. BITBOARDS

Un gran avenç a la dècada dels 70 va ser la introducció dels "bitboards" com a mètode per a representar el taulell. Aquest mètode ja s'havia aplicat a les dames perquè era més senzill.

La idea bàsica consisteix en utilitzar, en comptes d'un sol taulell, molts i que només continguin valors binaris, és a dir, bits (d'aquí el nom) que només continguin valors "true" o "false". Així tindrem un mínim de 12 bitboards; on un només tindrà el valor "cert" si en aquell quadre es troba la peça en qüestió. Per exemple el bitboard de "peons blancs" a l'inici de la partida i després d'haver efectuat el primer moviment 1. e4

F	F	F	F	F	F	F	F
T	T	T	T	T	T	T	T
F	F	F	F	F	F	F	F
F	F	F	F	F	F	F	F
F	F	F	F	F	F	F	F
F	F	F	F	F	F	F	F
F	F	F	F	F	F	F	F
F	F	F	F	F	F	F	F

F	F	F	F	F	F	F	F
T	T	T	F	T	T	T	T
F	F	F	F	F	F	F	F
F	F	F	T	F	F	F	F
F	F	F	F	F	F	F	F
F	F	F	F	F	F	F	F
F	F	F	F	F	F	F	F
F	F	F	F	F	F	F	F

D'aquesta manera amb operacions AND/OR es poden aconseguir molts més bitboards, tants com necessitats tinguem, com per exemple: superposant tots els bitboards de peces blanques amb portes "OR" podem obtenir el bitboard "peces blanques". Un altre necessitat que pot satisfer és saber quines caselles ataca cada peça en particular, i es podria aconseguir 64 bitboards, amb els noms "quadres que ataca la peça situada en e3".



ALGORISMES APLICATS ALS ESCACS: PROGRAMACIÓ D'UNA APLICACIÓ

Com hi ha tants bitboards com es pugui imaginar, aquests són els més utilitzats, però no els únics:

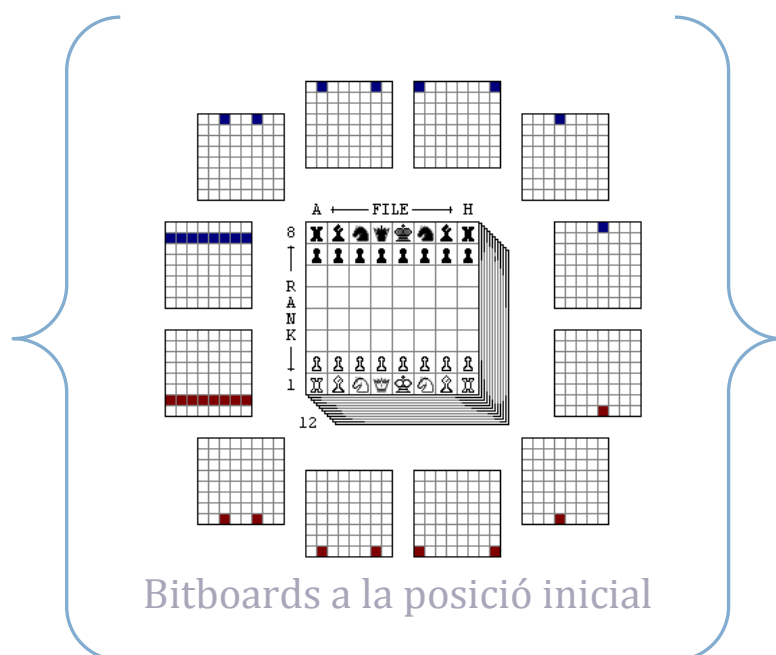
- 12 bitboards per a indicar la posició de les peces (1 per a avalls blancs, 1 per peons blancs, 1 per torres negres, etc.).
- 2 bitboards derivats dels anteriors on s'indica on estan les peces blanques i les negres.
- 64 bitboards que indiquen a quina posició ataca la peça situada a cada quadre (1 bitboard per quadre).
- 64 bitboards que fan la inversa de l'anterior: quins quadres ataquen a un determinat quadre (1 bitboard per quadre).

A més, el fet que el taulell tingui 64 caselles ajuda a l'hora de guardar els bitboards, el fet que 64 es pot expressar en la forma 2^6 , i que es pugui expressar en binari fàcilment ajuda molt a les màquines. Molts llenguatges de programació permeten crear contenidors de 64 bits de manera senzilla i ràpida.

Aquest mètode per emmagatzemar una posició és més complicat i cal que el programador o dissenyador de la màquina o programa sigui capaç de gestionar correctament

aquesta gran quantitat d'informació. Un dels mètodes per fer-ho és gestionant els bitboards amb arrays, però això torna a alentir el programa i no s'ha d'abusar.

Però un cop s'ha construït un programa amb bitboards, la velocitat que demostra tenir la CPU a l'hora de generar moviments és més gran i per això van marcar un abans i un després.





ALGORISMES APLICATS ALS ESCACS: PROGRAMACIÓ D'UNA APLICACIÓ

3.4. COMPARATIVA: AVANTATGES I INCONVENIENTS

Per tal que quedin aclarits els principals avantatges i inconvenients que suposa l'ús d'una determinada estructura he fet aquesta taula on queden reflectides les principals característiques.

Característiques	Llista de peces	Taula 8x8	Bitboards
Utilitzats des de	1930	1950	1970
Dificultat a l'hora del disseny	Fàcil	Moderat	Díficil
Necessitat de memòria	192 bits	De manera 100% eficient: 260 bits	12 bitboards: 798 bits. Més 64 bits per cada bitboard.
Predisposició a l'hora de generar moviments	Dolenta	Bona	Depèn del programador: generalment difícil.
Necessita espai extra per a informacions complementàries?	SÍ	NO	SÍ
Guarda similituds amb el joc real? (humanament parlant)	NO	SÍ	POQUES
Facilitat a l'hora de depurar (debug) problemes d'estructura	Fàcil	Moderadament fàcil	Díficil
Velocitat de tractament per la CPU	Molt alta	Lenta	Ràpida

3.5. QUÈ ÉS UN MOVIMENT?

Un moviment ha de contenir tres coses: la casella de sortida, la casella d'arribada i la possibilitat de coronació (si corona s'especificarà la peça). El que es fa és posar un 0 o "false" (depèn l'estructura que haguem escollit) a la casella de sortida i el valor de la peça o "true" allà on va la peça, eliminant els altres "true" que coincideixin amb aquella casella si utilitzem bitboards.



ALGORISMES APLICATS ALS ESCACS: PROGRAMACIÓ D'UNA APLICACIÓ

Una manera senzilla de representar això és utilitzar un enter de 5 xifres, on el primer número serà la peça a coronar, els dos següents la casella de sortida i els dos últims la d'arribada, si no corona, el número serà de 4 xifres. Si això es programés de manera eficient només caldria 15 bits, sis per a la sortida, sis per a la d'arribada (perquè amb 6 bits podem representar $2^6 = 64$ valors) i tres per a dir quina peça corona, perquè no fa falta saber si serà blanca o negra ja que podem mirar el peó d'origen.

D'aquesta manera, el moviment "1228" serà moure de la casella número 12 (d2) fins a la casella 28 (d4). Això és fàcil de fer utilitzant les operacions mòdul i divisió entera. El meu programa ho fa de la següent manera:

```
promoció := moviment / 10000 //primer número de "moviment"  
moviment := moviment - promoció*10000 //traiem el primer número  
casella_de_sortida := moviment / 100 //els dos primers números  
casella_d_arribada := moviment mod 100 //els dos últims números
```



4. ALGORITMES

Per tal de crear un programa que jugui a escacs s'ha de dividir la feina en diversos procediments per fer la programació més senzilla i clara. A aquesta tècnica de programació s'anomena *Divide and Conquer* (D&C). D'aquesta manera tindrem, a grans trets, quatre algorismes genèrics a tots els programes: el primer que ens dirà les jugades que es poden fer en una determinada posició, el segon farà una cerca en la posició i decidirà quina és la millor jugada basant-se en les avaluacions que farà el tercer algoritme i el darrer i més optatiu es tracta d'una acceleració en la cerca del segon algorisme i ho farà a través d'indexar posicions en una taula, de manera que ens podrem referir a una posició per un únic número.

4.1. GENERACIÓ DE MOVIMENTS

El primer algoritme ha de permetre al programa comprendre tots els moviments que pot fer. Primer haurà de crear els moviments "pseudolegals", és a dir, aquells moviments que segons les normes es podrien fer si no es deixés el rei propi indefens (el suïcidi no està permès en escacs) i després haurà de comprovar si aquests moviments pseudolegals són legals realment. Hi ha programes que aquesta segona comprovació la fan en mig de la cerca ja que produeix un gran alentiment del programa: ha de generar tots els moviments propis i tots els moviments del rival per a cada posició, i alhora comprovar si el propi rei es trobarà en escac. Com que aquest procediment és bàsic per a aprofundir en la posició més tard, ha de funcionar perfectament i no ha de donar lloc a errors, per tant aquesta serà una de les parts que més temps consumeixi.

També hi haurà moments pels quals serà útil tenir un segon generador de moviments que només crearà els moviments de captura o escac, ja que aquests moviments poden fer variar molt l'avaluació de la posició. Normalment aquest segon generador s'utilitza a la cerca de la quietud.

Històricament hi ha hagut tres maneres de generar els moviments i tenen molta relació amb la manera de tractar la cerca posterior:

- Generació selectiva: Els primers programes (del 1950 al 1970) pretenien només generar les jugades "bones" per tal de reduir el nombre de variants a investigar. Això feia que el treball que havia de fer el generador fos doble: generar tots els moviments i eliminar els dolents. A més els criteris que havia de seguir una màquina per trobar els possibles 5 millors moviments hauria d'eliminar línies bones. Aquesta manera de generar va fer-se inviable, ja que s'aconseguien millors resultats investigant l'arbre sencer.
- Generació progressiva: Aquesta tècnica pretén anar generant els moviments de manera progressiva, és a dir, primer un i comprovar la seva evolució dins de l'arbre, després un altre i mirar la seva evolució. Si es descartava el seguir avaluant una posició mitjançant la poda $\alpha\beta$ (mirar en "millores sobre Minimax"), no caldrà generar més



ALGORISMES APLICATS ALS ESCACS: PROGRAMACIÓ D'UNA APLICACIÓ

moviments. Aquest mode de generar moviments és molt útil per a màquines que no posseeixen hardware dedicat a la generació de moviments. A més, demostra que la màquina no sempre ha de generar tots els moviments. El principal problema que té és que si hagués de generar tots els moviments perdria més temps que el mateix algorisme però que els generés de cop. Tot i això, com que li passa poques vegades, aquest mètode s'ha demostrat el més viable.

- Generació completa: Quan els programes començaven a córrer en hardware dedicat específicament als escacs, una de les plaques que havia de tenir l'ordinador era per a la generació de tots els moviments que podia fer. D'aquesta manera els programes generaven tots els moviments de manera molt ràpida, accelerant alhora els càlculs en la cerca. Aquesta generació completa tornava els moviments legals (no "pseudolegals") ordenats segons l'ordre que es creies que seria més bona, per tant també ajudava a augmentar la velocitat de tractament de cada posició. Com a nota històrica, comentar que els programes de generació progressiva investigaven 200 posicions (o nodes) per segon, i BELLE, el primer ordinador amb hardware dedicat que utilitzava generació completa, investigava 120.000 posicions per segon.

Aquesta és la part que més alenteix el meu programa, tot i que la seva estructura en pseudocodi seria molt simple:



Algorisme Generador

Entrada: posició "P", un bit per a determinar si es generen els de blanques o de negres, un bit per a determinar si es generen moviments pseudolegals.

Sortida: una llista amb tots els moviments.

```
Llista "moviments"
Per a i = 0 fins a 63 fer:
  Si p[i] és alfil o dama
    Afegeix 4 diagonals a moviments
  fi si
  Si p[i] és una torre o dama
    Afegeix fila i columnes a moviments
  fi si
  si p[i] és cavall
    Afegeix salts de cavall a moviments
  fi si
  si p[i] és peó
    Afegeix moviments de peó
    si corona
      Afegeix totes les opcions de coronació
    fi si
  fi si
  si p[i] és rei
    Afegeix moviments de rei
  fi si
fi per a
si s'ha de comprovar que siguin legals
  per a cada moviment en llista
    fer_moviment en taulell auxiliar
    si el rei propi està en escac
      elimina moviment
    fi si
  fi per a
fi si
retorna moviments.
fi Algorisme
```

Els procediments per afegir els moviments de cada peça són relativament senzills, ja que els moviments ho són: diagonal, fila, columna, caselles al voltant de... El més complicat és el de generar el cavall, ja que no segueix una progressió estricta, però com a cada posició només té 8 opcions (poques) es pot introduir de manera manual. A més notar que en aquest pseudocodi no he utilitzat cap diferència per a blanques i per a negres, per tant aquest codi no funcionaria, s'hauria d'especificar si són peces negres o blanques.



ALGORISMES APLICATS ALS ESCACS: PROGRAMACIÓ D'UNA APLICACIÓ

Hi ha un altre problema que aquí no s'observa: posem per exemple que estem al quadre número 23 (a3) d'un taulell buit i aquí hi ha una torre. Per a generar les columnes sumarem 8 (o -8) a la casella inicial fins a arribar a fora del taulell (la casella de fora del taulell no la comptem), i les files sumarem 1 (o -1) fins arribar al límit, però, com sabem quan sortim del taulell? El meu programa calcula fila i columna de cada quadre i si és major que 7 parará de generar, però hi ha una altra manera molt interessant que cal considerar, les "mailbox" (o bústies, però tothom ho anomena en anglès), anomenades així per el seu impulsor Robert Hyatt i parteixen d'aquestes dues taules:

Mailbox mida 64								Mailbox mida 120										
21	22	23	24	25	26	27	28	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
31	32	33	34	35	36	37	38	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
41	42	43	44	45	46	47	48	-1	0	1	2	3	4	5	6	7	-1	-1
51	52	53	54	55	56	57	58	-1	8	9	10	11	12	13	14	15	-1	-1
61	62	63	64	65	66	67	68	-1	16	17	18	19	20	21	22	23	-1	-1
71	72	73	74	75	76	77	78	-1	24	25	26	27	28	29	30	31	-1	-1
81	82	83	84	85	86	87	88	-1	32	33	34	35	36	37	38	39	-1	-1
91	92	93	94	95	96	97	98	-1	40	41	42	43	44	45	46	47	-1	-1
								-1	48	49	50	51	52	53	54	55	-1	-1
								-1	56	57	58	59	60	61	62	63	-1	-1
								-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
								-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

La manera d'utilitzar-les és aquesta: estàvem al quadre 23 on hi ha una torre, accedim a la posició 23 de la mailbox de 64 elements, quin número conté? el 48. Anem a la mailbox de mida 120 i ens situem en la casella número 48, que conté un 23 (per a que ens sigui més fàcil d'entendre, si hagués estat positiva ja hauria estat bé). Ara imaginem que volem generar que la torre es mou una casella cap a la dreta (jugada il·legal perquè estaria fora del taulell), doncs mirem la casella de la dreta, la posició 49, de la taula "mailbox de 120", quin valor té aquí? -1, negatiu, és a dir està fora del taulell, aquí no es podrà moure, i parem de generar moviments en aquesta direcció.

Aquesta manera simplifica tant al generador de moviments com a l'hora de programar-lo. A més evita els errors de segmentació que es produirien si tinguéssim una torre a la posició 63 que volgués a anar a la 64, fora del taulell on no existeix.



Robert Hyatt, creador de Crafty i CrayBlitz, va utilitzar aquestes arrays per primer cop, anomenant-les "mailbox" per la seva similitud amb les bústies.



4.2. MINIMAX: CARACTERÍSTIQUES

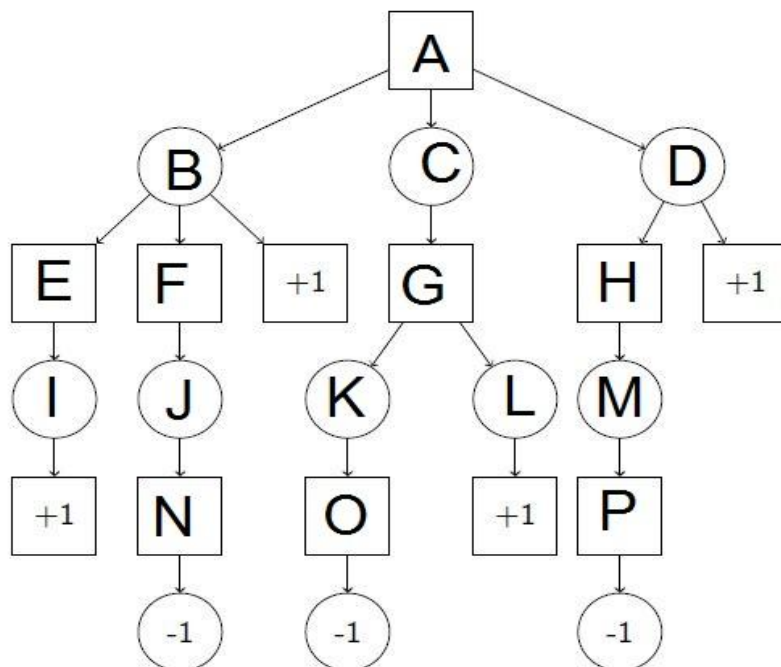
El minimax va aparèixer en primer lloc en el món de l'economia i de la teoria de jocs. Aquest primer teorema enunciat per John Von Neumann determinava que en els jocs de suma zero i d'informació perfecta com els escacs, dames, quatre en ratlla, entre altres, ens diu, de manera genèrica: "Cada jugador examinarà totes les estratègies possibles, i la seva jugada es basarà en minimitzar al màxim les pèrdues, suposant que el seu rival també vol minimitzar les seves pèrdues, d'aquesta manera s'obté l'estratègia òptima" o dit d'un altra manera, "La estratègia òptima s'obté quan els jugadors centren la seva estratègia en minimitzar la pèrdua màxima". Posteriorment, Claude Shannon l'aplicà als escacs amb l'objectiu de poder fer computable la cerca del moviment correcte, obtenint d'aquesta manera l'algoritme minimax.

Per tal de fer més clares les explicacions posteriors cal definir uns conceptes claus:

- Node: un registre que conté dades interessants i que es connecta amb uns altres nodes. En el cas dels escacs un node és una posició i s'uneixen a través de moviments que canvien la posició.
- Tipus de nodes:
 - Pare: Un node serà pare si aquest té enllaços amb altres nodes, i aquests altres nodes són resultat de modificar al pare.
 - Fill: cadascun dels nodes que resulta d'un node pare, un node fill pot ser alhora node pare d'altres nodes.
 - Arrel: Primer node, node que no té pare. En el cas dels escacs és la posició inicial o la posició en que es trobi el joc.
 - Terminal (o fulla): node que no té cap fill. En el cas dels escacs un node terminal ho pot ser perquè la partida s'hagi acabat (escac i mat, ofegat) o perquè s'ha arribat a un nivell de fondària adequat.
 - Branca: seguit de nodes que va des de l'arrel fins a un node terminal.
- Arbre de joc: conjunt format per molts nodes en forma d'arbre (entre dos nodes només hi ha un camí). Cada camí representa un moviment, una elecció de cada jugador.
- Fondària: nombre de nodes que té una branca.
- Amplada: nombre mitjà de nodes fills que hi ha per cada node pare.
- Algoritme de força bruta: es diu d'aquell algoritme que per tal d'arribar a la solució ha d'investigar totes les opcions.
- Per conveni anomenarem jugador "MAX" a aquell que vulgui maximitzar el valor de la posició i jugador "MIN" a aquell que vulgui minimitzar el valor de la posició. A més anomenarem nodes MAX a aquells que li toqui moure al jugador MAX, i nodes MIN a aquells que li toqui moure al jugador MIN.

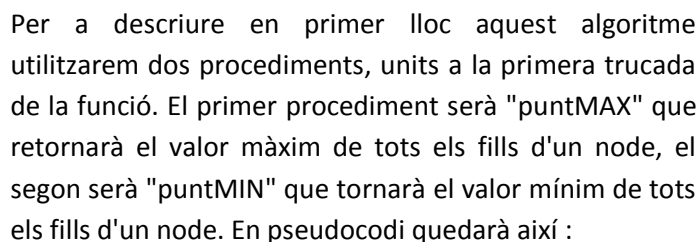


Per a explicar aquest algoritme recursiu de força bruta sortirem dels escacs i ho veurem de manera practica en un joc imaginari, molt més simple que té el següent arbre de joc:



Els quadrats representen nodes MAX, i tindran el valor que tingui el fill amb més valor, les rodones en canvi són nodes MIN i tindran el valor més petit que tingui el seus fills. Els nodes que ja tenen algun valor són nodes terminals, aquesta avaluació l'ha donat la funció d'avaluació, i com és un joc simple només tindrà dos estats finals: o guanya MAX (+1) o guanya MIN (-1).

Primer començarem per el node I, com que no té cap més elecció, aquest node tindrà el valor +1, a l'E li passa el mateix, per tant E valdrà +1. El mateix passa en la branca F-J-N, per tant F tindrà valor -1, el mateix que el fill d'N. El node MIN B haurà d'escollir entre E (+1), F (-1) i el que ja ha estat avaluat amb +1, escollirà el valor més petit d'aquests tres, i B escollirà el valor d'F, B valdrà -1. Seguim a investigar el valor de C, aquest serà el mateix que G, però G tindrà el valor més gran que tingui K i L. Com que a la branca K-O no hi ha més opció, K valdrà -1, i L valdrà +1 perquè tampoc hi ha eleccions. Així G escollirà entre K (-1) i L (+1), com és un node MAX escollirà aquell que maximitzi les seves opcions, doncs G escollirà L. Per tant C tindrà +1. L'última branca a investigar és la resultant de D. D haurà d'escollir entre el valor d'H i +1, com que el valor de la branca H-M-P no canviarà (no hi ha més opcions) H valdrà -1, i D escollirà la branca H-M-P. Recapitem: B valdrà -1 per la branca B-F-J-N, C valdrà +1 per la branca G-L i D valdrà -1 per la branca D-H-M-P. A escollirà el valor més gran d'aquests tres (A és node MAX) per tant s'escollirà el node C. L'arbre quedarà així:



Sortida: valor MAX de la posición

retorna avaluació de posició

per a cada moviment en moviments

```
puntuació = puntMIN(posició, profunditat-1)
```

```
desfermoviment(posició, moviment[i])
```

```
si puntuació > puntuació max
```

```
puntuació_max = puntuació
```

```
millor moviment = moviment[i]
```

fi si

fi per a

```

return puntuació max

```

fi si

fi funció

Sortida: valor MIN de la posición

retorna avaluació de posició

```
puntuació_min := +infinít
```

```
moviments := genera_moviments(posició)
```

per a cada moviment en moviments

```
fermoviment(posició, moviment[i])
```

```
puntuació = puntMAX(posició, profunditat-1)
```

```
desfermoviment(posició, moviment[i])
```

```
si puntuació < puntuació min
```

```
puntuació min = puntuació
```

```
millor moviment = moviment[i]
```

fi si

fi per a

retorna puntuació min

fi si

fi funció



ALGORISMES APLICATS ALS ESCACS: PROGRAMACIÓ D'UNA APLICACIÓ

La crida inicial en aquest cas no és important, però s'ha de fer si es vol fer que aquesta "maquinaria" funcioni. La crida inicial ha de ser "puntMAX(posició_arrel, profunditat_desitjada)". Noteu que la recursivitat és indirecta: puntMAX no es crida a si mateixa, sinó que crida a puntMIN que després cridarà a PuntMAX.

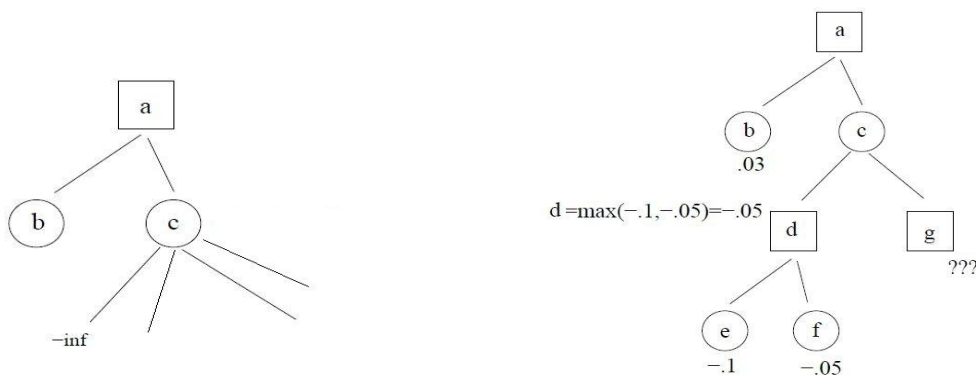
Per desgràcia aquest és un algoritme totalment inútil en els escacs, ja que ho ha d'investigar TOT, fins i tot aquelles branques que no tinguin sentit perquè no reporten cap millora, com la branca D-H-M-P de l'exemple anterior. A més es considera que l'arbre té un factor de ramificació de 35 (cada node pare tindrà 35 fills de mitja), d'aquesta manera si volem investigar a una profunditat de 6 plies (6 mitges jugades, 3 de blanques i 3 de negres) haurem que investigar $35^6 = 1838$ milions de posicions, i tenint en compte que la velocitat més gran que s'ha aconseguit en una màquina (DEEP BLUE) és de 200 milions de posicions per segon (notar que va ser dissenyada específicament per a que arribés a grans profunditats), trigaria 10 segons a arribar a una profunditat baixa. A més noteu que arribar al doble de profunditat, que ja seria un joc de GM, faria més de el doble de temps, ja que la relació és exponencial. Deep Blue trigaria 4 milions d'hores a arribar a aquesta fondària si utilitzés aquest algoritme.

Per aquest motiu hi ha moltes millores sobre el MiniMax, que tot seguit comentarem.

4.3. MILLORES SOBRE MINIMAX

4.3.1. PODA ALPHA BETA

La primera millora basa la seva idea en què si ja tenim una línia que ens proporciona una bona opció, no farà falta investigar més. En les següents figures es veu clarament:



Aquí no cal investigar tots els successors del node C MIN, ja que té la millor opció, -infinit. A més, A mai escollirà menys infinit si hi ha altres opcions, com ara "b".

En aquest cas no és tan clar, però té la mateixa lògica, no cal investigar g, perquè independentment del valor de G, C valdrà -0.5 o més petit, amb aquests valors el node MAX A mai escollirà C, escollirà B, que li maximitza més beneficis. Si C tingués més fills, aquests també serien podats.



Això no passaria si afegim els valor alpha i el valor beta: el valor α serà la màxima puntuació aconseguida fins al moment en els nodes MAX, i β serà la puntuació mínima aconseguida fins al moment en els nodes MIN. Si el node MIN pot aconseguir un valor més baix que la millor opció del rival (α) podarem aquest node perquè el rival no deixarà fer-la. El mateix passarà quan un node MAX pugui aconseguir un valor més alt que la millor opció del rival (β). La idea és que només haurem d'investigar l'interval que va des de α fins a β . En els moments en que es produeixi un interval del tipus $\alpha = 4$ i $\beta = 0$ (no hi ha cap valor entre mig), podarem aquest node, i alhora anirem actualitzant α i β al llarg de la cerca. Per tant s'ha de complir la condició $\alpha < \beta$, si això no passa, podarem el node.

Per tal d'explorar les condicions de poda, hem d'analitzar diverses suposicions:

- Imaginem que estem en un node MAX i obtenim el valor d'un fill, que anomenarem valor_i . Si aquest valor és més gran (estrictament) que α , actualitzarem el valor α ($\alpha := \text{valor}_i$) i continuarem investigant. Si a més $\text{valor}_i \geq \beta$, ens trencarà la condició $\alpha < \beta$ (perquè també coincidirà la condició anterior i s'actualitzarà el valor α) i per tant retornarem α , per tal que el node MIN no agafi mai aquest valor. Noteu que en aquest cas és quan es verifica $\alpha > \beta$, i tornem el valor més gran, α , per a que el següent node MIN no l'agafi.
- Imaginem que estem en un node MIN i obtenim el valor d'un fill, que anomenarem valor_i . Si aquest valor és més petit (estrictament) que β , actualitzarem el valor de β ($\beta := \text{valor}_i$) i continuarem investigant. Si també es verifica que $\text{valor}_i \leq \alpha$, és a dir, la millor opció que té MIN la pot evitar MAX en algun lloc abans de l'arbre, en aquest cas deixarem d'investigar i retornarem β per tal que MAX no agafi aquest valor. Recordeu que en aquest moment s'incompleix que $\alpha < \beta$, és més, ha de ser $\alpha > \beta$ per a que passi, i tornem el valor més petit, β , per a que MAX no l'agafi.

Aquest seria el pseudocodi de les anteriors funcions "puntMAX" i "puntMIN" amb les podes $\alpha\beta$:



ALGORISMES APLICATS ALS ESCACS: PROGRAMACIÓ D'UNA APLICACIÓ

Funció puntMAX

Entrades: Posició "p", enter " α ", enter " β ", enter "prof"

Sortida: enter, valor MAX d'aquesta posició.

```
Si és node fulla(p) o prof == 0
    retorna avaluació(p)
sinó
    llista moviments := genera_moviments(p)
    per a cada moviment en moviments
        fermoviment(moviment, p)
        puntuació := puntMIN(p,  $\alpha$ ,  $\beta$ , prof-1)
        desfermoviment(moviment, p)
        si puntuació >  $\alpha$ 
             $\alpha$  := puntuació
            si puntuació >  $\beta$ 
                retorna  $\alpha$ 
            fi si
        millor_moviment := moviment
    fi si
    fi per a
    retorna  $\alpha$ 
fi funció
```

Funció puntMIN

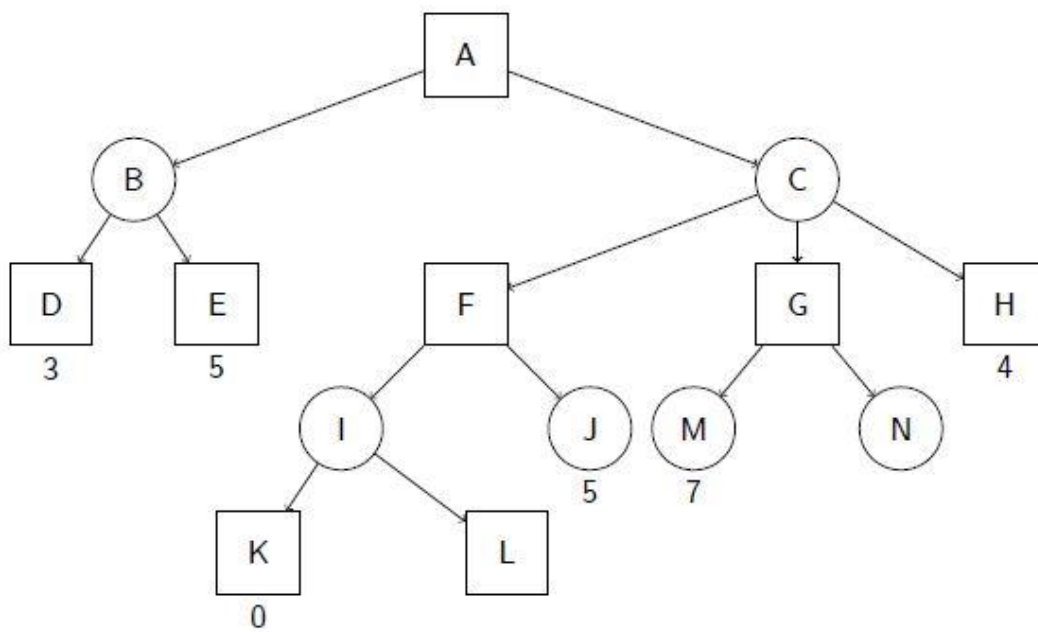
Entrades: Posició "p", enter " α ", enter " β ", enter "prof"

```
Si és node terminal(p)
    retorna avaluació(p)
Sinó
    llista moviments := genera_moviments(p)
    per a cada moviment en moviments
        fermovimet(moviment, p)
        puntuació := puntMAX(p,  $\alpha$ ,  $\beta$ , prof-1)
        desfermoviment(moviment, p)
        si puntuació <  $\beta$ 
             $\beta$  := puntuació
            si puntuació <  $\alpha$ 
                retorna  $\beta$ 
            fi si
        millor_moviment := moviment
    fi si
    fi per a
    retorna  $\beta$ 
fi funció
```

En aquest cas la primera crida a la funció és important perquè determina els primers valors d' α i de β , si no ens volem arriscar, el que podem fer és cridar "puntMAX(p, -infinit, +infinit, profunditat_desitjada)" i la funció mateixa s'autoregularà.



Per tal de veure que aquest algoritme funciona prosseguim a investigar un arbre d'un joc imaginari, igual que hem fet amb el Minimax. La figura de la que partim és aquesta:



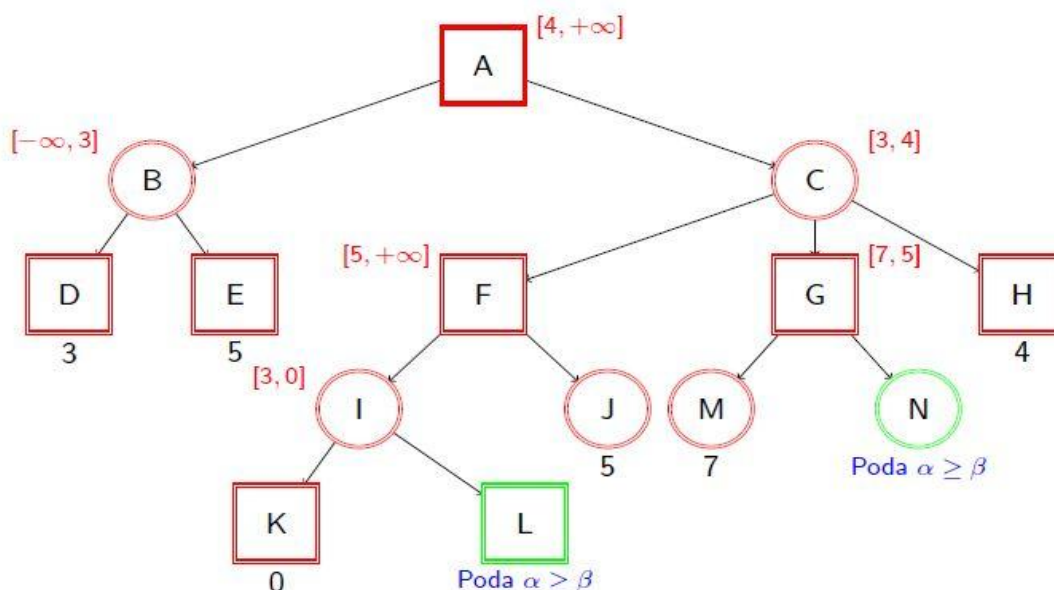
Ho examinem de manera seqüencial perquè l'ordre importa i molt:

1. Posem $\alpha = -\text{infinít}$, $\beta = +\text{infinít}$ en els nodes A, i el primer a investigar, B.
2. El node MIN B actualitzarà beta a 3, ja que $\text{valor}_d < \beta$.
3. S'investiga el node E, com que no ens dóna millores, no fem res.
4. El node B retorna 3, actualitzem α del node A, perquè $\text{valor}_b > \alpha$.
5. Transmetem $\alpha = 3$ i $\beta = +\text{infinít}$ a la branca C-F-I.
6. Des del punt de vista de I, $\text{valor}_k < \alpha$ i $\text{valor}_k < \beta$, traiem (podem) els fills restants de I, en aquest cas només L, I retorna el valor beta, 0.
7. F continua investigant, I no li provoca res, F continua tenint $\alpha = 3$ i $\beta = +\text{infinít}$. El $\text{valor}_j > \alpha$, s'actualitza α .
8. valor_f , per al seu pare C, és 5. Com que $5 < \beta$, actualitzem β . En el node C, $\alpha = 3$, $\beta = 5$.
9. Investiguem G, que té la mateixa α i β que C. El $\text{valor}_m < \beta$ i més petit que α , causa poda, no investigarem els nodes restants fills de N. G retorna α , que val 7.
10. A C aquest 7 no li canvia res, continua investigant. El valor del següent node és 4, com que $\text{valor}_h < \beta$, s'actualitza beta, però és més gran que α , per tant no es poda.
11. C ha acabat d'investigar i retorna β , 4. El node A actualitza el valor α i acaba d'investigar (no hi ha més moviments).

Per tant la seqüència òptima és A-C-H.



Figura final:



Que hauria passat si primer haguéssim investigat aquesta branca A-C-H? Que la quantitat de podes que hauria fet hauria estat més gran, per això té gran importància l'apartat "Ordenació de jugades" que tractarem més endavant. Aquest tipus d'algorisme s'anomenen *Best-First*, funcionen millor si la primera opció és la correcta.

Però quins avantatges, numèricament parlant, ens reporta l'ús d' $\alpha\beta$ respecte al MiniMax? La resposta és "depèn de l'ordenació": en un arbre amb les jugades perfectament ordenades es redueix de n nodes de l'arbre investigats a \sqrt{n} nodes. I si la ordenació és dolenta, la pitjor possible? El algoritme no podarà res i trigarà el mateix que el MiniMax, és a dir, només presenta guanys respecte al MiniMax.

4.3.2. NEGAMAX

Negamax és un algoritme derivat del MiniMax que obté els mateixos resultats pel mateix cost computacional. La millora és la claredat a l'hora de veure l'algoritme, en comptes d'utilitzar dues funcions, puntMax i puntMin, utilitza una sola funció, Negamax. Es basa en la igualtat: $\text{puntMax}(\alpha, \beta) = -\text{puntMin}(-\beta, -\alpha)$. Així a la línia on puntMax crida a puntMin, la substituïrem per aquesta altre:

Puntuació := -negamax(p, - β , - α , prof-1)

D'altra banda s'ha de tenir en compte que l'avaluació de la posició està determinada per a qui té el torn, és a dir, la funció d'avaluació calcularà el valor numèric de la posició en funció de a qui li toca moure. Això és senzill ja que només és canviar el signe de l'avaluació: una posició que té +5 per a les blanques té -5 per a les negres.



4.3.3. FER PETIT L'INTERVAL DE CERCA

L'interval $[\alpha, \beta]$ ¹ és molt important, si aquest interval és molt petit vol dir que el programa està a prop de la solució, si és molt gran és que està lluny. Per tant ens interessarà fer-lo el més petit possible, per a fer-ho i ha dues tècniques molt útils:

- Jugada nul·la (*Null move*): Parteix de la lògica “Segur que hi ha alguna jugada millor que no fer res”. Primer es fa una petita cerca d'una profunditat inferior (3 o 4 jugades) donant-li el torn al rival, és a dir, es fa una jugada nula. El resultat d'aquesta cerca serà el primer valor d'alpha, així, les línies que no reportin millores no seran investigades. El defecte que té aquesta idea és que hi ha moments en els escacs que un jugador perd pel fet que ha de moure (*Zugzwang*) i per tant desacreditaria la idea inicial.
- Un altre idea és cridar a la funció d'avaluació i demanar-li el valor numèric que li dona a la posició inicial, i aquesta seria la puntuació esperada però com no és exacte a la funció AlphaBeta li passarem que α és aquesta puntuació esperada menys el valor de dos o tres peons i β serà la puntuació esperada més dos o tres peons. Si en comptes de passar dos o tres peons de marge passéssim més marge no podríem tant però aniríem més segurs, en canvi si passéssim menys marge podríem sofrir que el vertader valor de la posició estigués fora d'aquest marge i per tant la cerca no funcionés. Aquest mètode no es pot aplicar si la posició en que es troba la partida ha estat el resultat d'un canvi de peces, per exemple: si el jugador MIN ha matat la dama del rival i el jugador MAX la recuperar en la següent jugada, ja que l'avaluació de la funció es basa fortament en el material de cada bàndol.

Aquests mètodes no s'han d'utilitzar al mig de la cerca ja que els seus defectes s'agregarien: els programes analitzen milions de posicions per segon, segur que en una d'aquestes es compleix el que fa falta per a que aquests mètodes no valguin. Si algun d'aquests intents de fer l'interval més petit fallés (la millor jugada no està dintre de l'interval), s'hauria de repetir la cerca.

4.3.4. ORDENAR LES JUGADES

Un procediment senzill per tal d'accelerar els càlculs es tracta d'ordenar les jugades. Ja em vist que si les jugades s'ordenen perfectament es redueix notablement el número de posicions a analitzar. Els algoritmes d'ordenació estan molt estudiats i per això la majoria de llenguatges de programació permeten aquests algoritmes de manera prefabricada, només els has de dir quin són els criteris per a decidir quin un moviment va abans d'un altre. Així es demana una

¹ A aquest interval s'anomena *Aspiration Window*, he preferit la traducció “interval”.



ALGORISMES APLICATS ALS ESCACS: PROGRAMACIÓ D'UNA APLICACIÓ

funció que determini de dos moviments, quin serà al que anirà primer i el propi algoritme farà la resta.

El meu programa utilitza moltes maneres de “desempatar” entre dos moviments: primer s’investiga una cosa, si són iguals es passa a investigar la segona, si a més són iguals en aquest segon criteri, es mirarà un tercer... Aquests criteris posats en ordre són:

1. Si la posició actual ja existeix en el “hash”, és a dir, si aquesta posició ja s’ha investigat, tindran preferència l’últim millor moviment, és a dir, l’últim que hagi fet el tall.
2. Si un moviment comporta una coronació, és a dir, la promoció d’un peó a un altre peça, aquest tindrà preferència. Si els dos moviments coronen tindrà preferència el que coroni una peça de més valor. Si a més coronen la mateixa peça es dona preferència de forma arbitrària.
3. En tercer lloc s’implementa la idea MVV/MLA (*Most Valuable Victim/ Least Valuable Atacker*, literalment, “Víctima més valuosa/Atacant menys valuós”), consisteix a donar preferència a aquelles jugades que guanyin el major material possible amb el menor cost possible. Es parteix que Rei(R) > Dama(D) > Torre(T) > Alfil(A) ≥ Cavall(C) > Peó(P), per tant primer s’investigarà moviments del tipus PxD, PxD, PxA, PxC, CxD..., després els moviments que no guanyen ni perden, CxC, AxA, AxC, DxD..., i per últim, els moviments que suposen perdre material TxA, CxP, DxD... Segons això es pot fer aquesta taula: les columnes indiquen la peça que es matarà i la fila la peça que mata. El número que conté la intersecció és la prioritat que tindrà a l’hora d’ordenar-se.

Mata/Es matat	No mata	P	C	A	T	D
P	-1	0	1	2	3	4
C	-2	-1	0	1	2	3
A	-3	-2	-1	0	1	2
T	-4	-3	-2	-1	0	1
D	-5	-4	-3	-2	-1	0
R	-6	-5	-4	-3	-2	-1

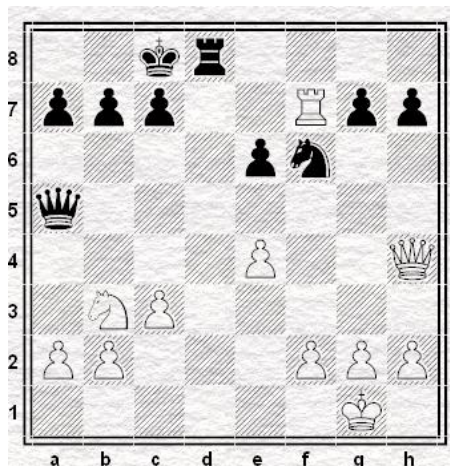
4. Si tot i així coincideixen, es dóna preferència a que es mogui la peça menor valor.
5. Si encara coincideixen es considera millor moure una peça cap al centre, es considera centre del taulell la casella 27 (e4).

Si tot i així coincideixen aquests valors es dona preferència de manera pseudo-arbitrària, anirà primer aquell moviment que segons el seu número enter sigui més gran (veure “Què és un moviment?”).



ALGORISMES APLICATS ALS ESCACS: PROGRAMACIÓ D'UNA APLICACIÓ

Una altre tècnica heurística és l'ús de les *Killer moves* que són aquelles jugades que en el mateix nivell de fondària han causat l'últim tall. Per exemple: en la següent posició imaginem que el programa investiga a 2 plies de fondària. La millor jugada del blanc en primer lloc seria fer CxD, guanyant material. Però les negres donarien escac i mat amb Td1++. Doncs la primera jugada que s'ha d'investigar en el segon nivell de fondària és Td1, és a dir, ha de tenir preferència per davant de les altres. És a dir, quan el programa investigui Txg7 o Txf6, la primera resposta que li vindrà serà Td1.



Aquesta idea no la utilitza el meu programa ja que el *hash* que té és molt gran i aquesta petita aproximació no ajuda gaire.

Per fer veure que l'ordre és important, aquí teniu dues taules de la mateixa cerca d'un node MIN, amb les jugades ven ordenades o sense ordenar (penseu que per a saber el "valor node" s'ha de fer una cerca, si posa "podat" significa que no ha investigat tots els fills):

Sense ordenar		
Ordre	Valor del node	α
1	9	9
2	7	7
3	5	5
4	8 (podat)	5
5	2	2
6	2 (podat)	2

Ordenat		
Ordre	Valor del node	α
1	2	2
2	2 (podat)	2
3	5 (podat)	2
4	7 (podat)	2
5	8 (podat)	2
6	9 (podat)	2

4.3.5. PROFUNDITZACIÓ ITERATIVA

La profundització iterativa (*Iterative deeping*) consisteix a fer n cerques per a arribar al nivell de fondària n: primer investigarà a una fondària 1, després una fondària 2, després una fondària 3... fins a arribar a la fondària n. En principi aquest mètode va contra la lògica humana, però ho fem per tres motius:



1. Per arribar a un nivell n de profunditat fa falta més temps que el sumatori de tots els temps de fondària anterior, per tant, no es perd temps, s'aprofita d'una altra manera.
2. Si ens demanen que fem la jugada en un temps determinat, sempre tindrem una jugada a punt, no haurem d'acabar d'investigar el nivell ni deixar-lo a mitges, només utilitzarem el resultat de l'última iteració.
3. Per omplir el *hash*, una petita base amb totes les posicions investigades. D'aquesta manera utilitzem els resultats de les cerques anteriors per a ordenar millor les jugades i fer la cerca profunditat n en menys temps.
4. També podem utilitzar aquestes cerques com a primers valors d' α i β , més i menys un marge d'error de tres peons, igual que s'ha comentat a l'apartat "fer petit l'interval de cerca", però en comptes d'utilitzar els valors que et dona la funció d'avaluació, utilitzar els valors de la cerca en un nivell inferior.

4.3.6. EVITAR L'EFECTE HORITZÓ: LA CERCA DE LA QUIETUD

L'efecte horitzó és el pitjor problema que presenta l'algoritme Minimax en els escacs, a més s'ha de dir que aquest problema és irresoluble en el Minimax. L'efecte apareix perquè el programa no veu més enllà dels nivells de profunditat en què es troba. Per exemple: un programa que investiga a 7 moviments de fondària i al moviment número 7 el programa calcula que captura la dama rival, és a dir, que rep una valoració positiva en una línia on guanyarà dama. Si per a fer-ho ha d'entregar peça, ho farà sense dubtar. Però el programa no sap si al moviment 8 el rival tornarà capturar la dama i per tant el sacrifici no és bo, o si el sacrifici és bo i guanyarà veritablement la dama.

La manera d'intentar resoldre aquest problema és que en comptes de passar la posició a avaluar quan s'arriba al nivell de fondària desitjat es passa a un altre funció que investigará més a fons totes les captures de material, les coronacions i els escacs, ja que aquests són els factors tàctics que poden fer que l'avaluació estàtica de la posició no valgui.

Aquesta segona funció s'anomena *quiescence search* o cerca de la quietud. La segona cerca serà igual que la primera (Negamax amb podes alpha-beta) però en el seu generador de moviments només generarà els moviments que puguin canviar amplament la valoració estàtica de la posició, com els esmentats anteriorment. Quan el programa consideri que està suficientment a fons o que la posició ja és "estàtica", cridarà a la funció d'avaluació perquè retorni el valor de la posició.

Aquesta cerca de la quietud es pot fer a més fondària ja que el factor de ramificació es redueix considerablement: en una posició "normal" hi ha entre 30 i 40 moviments entre els quals només 5 són de captura, escac o coronació. Això vol dir que per a arribar a una profunditat de 5 nivells, s'han d'investigar $5^5 = 3125$ branques, mentre que a la cerca normal només podria arribar a 2 nivells de fondària.



La cerca de la quietud no és perfecte però és necessària perquè elimina el 80% dels errors derivats del efecte horitzó. Noteu que aquesta “solució” només aplaça al problema a una jugada més llunyana, no el soluciona. Hi ha un exemple famós en què Ponomarev va jugar contra el programa comercial “Fritz” al 2005 i el programa va perdre perquè no va veure una línia a 10 jugades (10 moviments blancs i 10 negres).

4.4. AVALUACIÓ ESTÀTICA DE LA POSICIÓ

La part d'avaluar numèricament una posició respecte els avantatges que presenta per a un i altre bàndol és la part més personal de cada programa, ja que es prioritzaran uns avantatges o uns altres depenent dels coneixements sobre els escacs de cada programador. Però totes les funcions d'avaluació han de buscar l'equilibri entre el coneixement dels escacs i la velocitat d'execució. Una funció d'avaluació exhaustiva, a nivell de GM (*Grand Master*), trigaria massa en profunditzar i investigaria menys posicions, en canvi, una avaluació superficial basant-se en la posició de les peces pot portar a cometre errors fatals i a que l'arbre no tingui sentit.

Hi ha moltes maneres de dissenyar aquesta part i encara no s'ha trobat la més eficient ni la més perfecta. En aquest aspecte treballen tant programadors com GM, per això el que explicaré a continuació és només la manera que té el meu programa a l'hora d'avaluar la posició, no totes les que es poden fer.

A la meua funció d'avaluació hi ha una variable “score”, que s'inicialitza a 0, i se li sumen els valors positius si el jugador blanc té avantatge en un aspecte o si el negre té desavantatge en un altre i se li sumen valors negatius per als desavantatges blancs o per a els avantatges negres. A l'acabar totes les sumes i restes es retorna aquest valor.

El primer que té en compte, ja que des d'un punt de vista computacional és senzill, és la posició de les peces. Per a fer això s'utilitzen les arrays PC/SQ, de les seves sigles angleses *Piece/Square*. Hi ha una array diferent per a cada peça, on cadascuna conté la puntuació extra que té una peça pel fet d'estar a un determinat quadre (per exemple a les arrays de la taula següent es dóna una puntuació de -20 si un cavall es troba a la casella 0). A més, hi ha una matriu extra que dóna avantatge a les peces que es troben en el centre, sigui quina sigui. Cal destacar que el rei té dues arrays, una destinada per als finals, per tant, es premiarà al rei que estigui al centre, i una altra destinada a tot el joc, on es castigarà el rei mal situat, és a dir, no enrocat. Per a decidir quina em d'utilitzar ens basem en el número de peces que hi ha sobre el taulell. Per últim, per no fer matrius per duplicat, per a blanques i per a negres, hi ha una taula indexades per quadres des del punt de vista del negre i conté el quadre que correspondria des del punt de vista del blanc (el quadre 63 negre correspon al 0 blanc, el 62 a l'1, el 60 al 2...). S'utilitzaria així: `matriu_bonus[auxiliar[quadre]]`. Els claudàtors (“[]”) es llegeixen com “en la posició”. Aquestes són les matrius que utilitzen el meu programa (recordeu que això en realitat està en fila, aquí estan posades com si fos un taulell, amb les blanques a dalt):



ALGORISMES APLICATS ALS ESCACS: PROGRAMACIÓ D'UNA APLICACIÓ

0	1	3	5	5	3	1	0
1	2	5	8	8	5	2	1
3	5	8	10	10	8	5	3
5	8	10	15	15	10	8	5
5	8	10	15	15	10	8	5
3	5	8	10	10	8	5	3
1	2	5	8	8	5	2	1
0	1	3	5	5	3	1	0

-20	-10	-10	-10	-10	-10	-10	-20
-10	-5	-5	0	0	-5	-5	-10
-5	0	10	5	5	10	0	-5
-5	5	7	15	15	7	5	-5
-5	5	7	15	15	7	5	-5
-5	7	7	7	7	7	7	-5
-10	-5	-5	0	0	-5	-5	-10
-20	-10	-10	-10	-10	-10	-10	-20

Matriu "centre"

Matriu "cavalls"

0	0	0	0	0	0	0	0
0	0	-2	-5	-5	-2	0	0
2	2	2	2	2	2	2	2
5	5	5	5	5	5	5	5
10	10	10	10	10	10	10	10
25	25	25	25	25	25	25	25
50	50	50	50	50	50	50	50
0	0	0	0	0	0	0	0

0	20	-5	-10	-5	10	20	-10
-10	-10	-10	-10	-10	-10	-10	-10
-30	-30	-30	-30	-30	-30	-30	-30
-35	-35	-35	-35	-35	-35	-35	-35
-50	-50	-50	-50	-50	-50	-50	-50
-65	-65	-65	-65	-65	-65	-65	-65
-80	-80	-80	-80	-80	-80	-80	-80
-80	-80	-80	-80	-80	-80	-80	-80

Matriu "peons"

Matriu "rei al mig joc"

0	10	10	10	10	10	10	0
5	15	15	15	15	15	15	5
10	20	20	25	25	20	20	10
15	30	35	40	40	35	30	15
15	30	35	40	40	35	30	15
10	20	20	25	25	20	20	10
5	15	15	15	15	15	15	5
0	10	10	10	10	10	10	0

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

Matriu "rei al final"

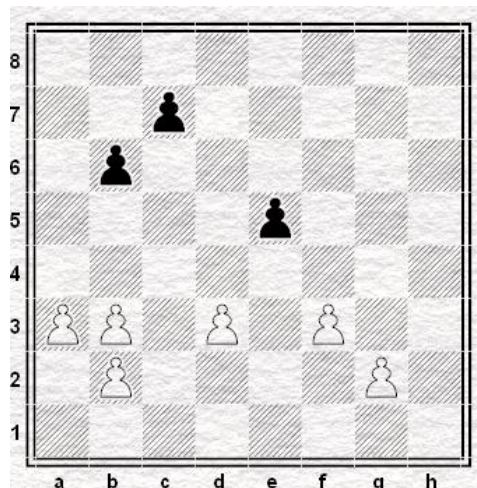
Matriu "auxiliar"



ALGORISMES APLICATS ALS ESCACS: PROGRAMACIÓ D'UNA APLICACIÓ

La funció un cop ha avaluat les PC/SQ, observa l'estructura de peons i segons això determina tres avantatges:

- Peons passats: són aquells que no tenen cap peó rival davant ni a les columnes adjacents. La importància d'aquests peons és que si volen coronar no tindran cap impediment. El fet de tenir un peó passat comporta un "bonus" de 20 punts.
- Peons aïllats: són aquells que no tenen un peó a les columnes adjacents. Aquests peons solen ser dèbils ja que no poden ser protegits per un altre peó. El fet de tenir un peó aïllat comporta -10 punts.
- Peons doblats: quan un peó captura una peça i es posa davant d'un altre peó, es creen dos peons doblats. Aquests peons són inútils, es molesten l'un a l'altre contínuament. Si hi ha dos peons doblats s'aplica un "bonus" de -14, si n'hi ha tres a la mateixa columna de -28.



Els peons de b2 i b3 estan doblats.

El peó de d3 està aïllat.

Els peó de g2 està passat

Un altre aspecte que tindrà en compte l'avaluació és el material. Les peces, pel fet d'estar al taulell, comporten una avantatge. Aquesta part de l'avaluació té més pes en el resultat final. El valor de les peces que dona el meu programa és:

- Peó ♙ : 100 punts
- Alfíl ♗ : 325 punts
- Dama ♕ : 900 punts
- Cavall ♘ : 300 punts
- Torre ♖ : 500 punts

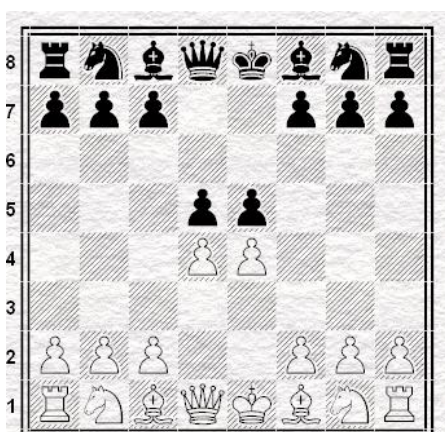
Tot i que a l'hora de jugar, un cavall i un alfíl valen el mateix, també és veritat que la parella d'alfíls és més forta que la parella de cavalls. D'aquí la superioritat del valor de l'alfíl per davant del cavall.

Per últim i el que més triga és el càlcul de la mobilitat de les peces més fortes: alfíls, torres i dames. El que fa el programa per veure la mobilitat de cada peça és generar tots els moviments pseudo-legals (mirar "Generació de moviments"). Cada moviment en la llista resultant es multiplica per un determinat factor. En el cas dels alfíls es puntua 2 punts per cada casella. A les torres es multiplica per 2 el nombre de caselles on pot anar, però es comença des de -6, és a dir, que per a que una torre no compti, ha de tenir un mínim de tres caselles. A la dama li donem 1 punt per cada quadre on es pugui moure, i com a extra també li sumem la seva posició en la PC/SQ "centre" menys cinc.



El procediment d'avaluació és cridat un cop s'ha determinat que el node és terminal perquè no es pot aprofundir més, no perquè la posició sigui escac i mat o ofegat. Si la posició és d'escac i mat s'avalua de manera indirecta per l'algoritme de cerca perquè és important saber quan es dona l'escac i mat per a evitar que el programa se suïcidi: si veu que l'escac i mat és inevitable l'ha d'allargar el màxim possible. Això és més fàcil de fer a l'algoritme de cerca: quan diu "és node terminal" calcula aquests matisos.

4.5. TAULES DE TRANSPOSICIÓ: EL *HASH*



En els escacs és molt fàcil que es donin dues posicions idèntiques en nodes molt llunyans dins de l'arbre. Per exemple a la posició de la figura es pot arribar per quatre branques: 1. e4 e5 2. d4 d5 o 1. e4 d5 2. d4 e5 o 1. d4 d5 2. e4 e5 o 1. d4 e5 2. e4 d5. Quan passa això en els escacs es diu que una línia *transposa* a un altre. En aquests casos no és interessant seguir cercant amb minimax els quatre nodes, ni tampoc avaluar-la quatre vegades amb la funció d'avaluació. Per tal de no repetir-ho es recorre a les taules *hash*.

Una taula *hash* és una forma d'emmagatzemar dades molt utilitzada. Consisteix en que a través d'unes claus, *keys*, s'accedeix a uns registres determinats. Per exemple, l'array que s'utilitza per a representar el taulell és una taula *hash*, a partir d'un número enter entre 0 i 63 s'accedeix al valor del quadre, que serà un altre enter. En aquest cas la clau és el número del quadre i el registre o contingut és un valor codificat en forma de número que nosaltres hem determinat que serà una peça o un buit.

El que volem és una taula indexada segons una posició que contingui diverses dades interessants, com l'últim valor que tenia aquell node, l'última millor jugada o a quina profunditat s'han trobat aquestes dades. Això accelera el càlcul per dos motius: per una banda s'aconsegueixen majors podes $\alpha\beta$ perquè les jugades s'ordenen millor i per una altra el programa fa menys "feina" quan pot reutilitzar dades del passat.

El primer problema és que no podem guardar tota la posició dins la taula ja que ocuparia molta memòria ràpidament per a poques posicions, penseu que si s'utilitzessin 8 bytes per cada posició es gastarien 8 MB per a emmagatzemar un milió de posicions i hi ha programes que investiguen aquest nombre de posicions cada segon. És a dir, s'ha d'utilitzar un altre mètode per a generar la clau d'una posició qualsevol. El mètode que més s'utilitza va ser proposat per Albert L. Zobrist al 1970 i es pot aplicar tant als escacs, Othello, Go, dames... a tots aquells jocs que tinguin un taulell abstracte per a representar els moviments.



ALGORISMES APLICATS ALS ESCACS: PROGRAMACIÓ D'UNA APLICACIÓ

El procediment de *hashing* de Zobrist aplicat als escacs segueix aquests passos:

1. Es crea una matriu de 12x64, 12 perquè és el número de peces que hi ha (6 blanques i 6 negres) i 64 perquè és el número de caselles. Es col·loquen números al atzar, tant és quin número sigui però no ha de canviar durant l'execució del programa. Els números han de tenir una llargada de bits predeterminada, en el cas del meu programa és de 64 bits.
2. En una variable, anomenem-la "key", la inicialitzem a un valor predeterminat, per exemple 0.
3. Anem quadre per quadre (del 0 al 63) i si trobem una peça fem *key* XOR amb el número que correspon de la primera matriu que hem creat, a fila "peça que hem trobat" i columna "casella". Un cop hem acabat, fem *key* XOR 1 si li toca al blanc o *key* XOR 0 si li toca al negre.

Un cop fet això tindrem aquesta variable "key" que serà la clau de la taula o de les taules *hash* (podem crear tantes taules *hash* com faci falta) on guardarem les informacions que necessitem.

Normalment la llargada en bits dels números generats a l'atzar és important perquè també serà la llargada de la *key*, i el *hash* es veurà reduït a tenir un número determinat de posicions. Per exemple si generem números al atzar de 8 bits la taula *hash* haurà de tenir $2^8 = 256$ entrades diferents. En el cas del meu programa no té 2^{64} entrades on cada entrada ocuparia 2 bytes, això ocuparia massa memòria. El que fa és aprofitar un dels contenidors que proporciona les biblioteques STL del C++, el "*map*". Un *map* permet, entre altres coses, que existeixi la clau número 7 sense que existeixi l'entrada número 6 o qualsevol altre. És a dir, en un *map* només existeixen aquelles entrades que ens seran útils.

Aquest mètode es basa en l'operació lògica XOR. XOR és una operació que es fa bit a bit de dos números binaris. El resultat serà cert quan una, i només una de les seves entrades, sigui certa. Utilitzem XOR ja que la seva naturalesa fa que ella mateixa sigui la seva inversa: al fer XOR al mateix número dues vegades el número es queda tal com estava al principi. Per exemple si agafem el número 11010010_{bin} i el 01011000_{bin} el resultat de fer XOR d'aquests dos números serà 10001010_{bin} i si aquest resultat li tornem a XOR amb el segon número obtindrem el primer un altre cop. Noteu que res té a veure amb cap operació matemàtica dels nombres en decimal ($210 \text{ XOR } 88 = 138$ i $138 \text{ XOR } 88 = 210$). Doncs és cert que si a i b són nombres enters $a \text{ XOR } b \text{ XOR } b = a$.

El gran avantatge que presenta aquest mètode de *hashing* és que si tenim una posició qualsevol i fem un moviment, no haurem tornar a mirar la posició de totes les peces un altre cop, podem aprofitar la naturalesa del XOR explicada abans. Imaginem que movem una torre negra que està al quadre 27 (e4) fins a al quadre 11 (e2) i mata al peó que hi havia allà. Haurem de fer només 4 operacions de XOR:



ALGORISMES APLICATS ALS ESCACS: PROGRAMACIÓ D'UNA APLICACIÓ

```
key := key XOR matriu[torre_negra][e4] XOR matriu[torre_negra][e2] XOR  
[peó_blanc][e2] XOR canvi de torn //Anomeno matriu a la primera matriu de  
12x64 generada a l'atzar.
```

El primer XOR fa desaparèixer a la torre negra de e4 de la *key*, perquè abans ja estava “XORreada” dins la *key* original i al torna-la a ficar desapareix. El segon XOR fa aparèixer una torre negra en el quadre e2, perquè abans no estava dins la *key*. El tercer fa desaparèixer el peó blanc que s’ha matat i l’últim XOR és per a que es canviï el torn.

Un dubte que podria sorgir en aquest moment és el perquè utilitzem XOR i no sumes i restes o multiplicacions o divisions. En primer lloc, la velocitat en què una màquina fa les operacions lògiques, com XOR, és molt més ràpida que les altres. I segon el nombre de col·lisions, és a dir, que dues claus siguin iguals per a posicions diferents, és molt més baix. També cal dir que si hi hagués una col·lisió de claus no provocaria grans mals al programa.

La quantitat de taules de *hash* que fan falta van determinades per les necessitats que van sorgint. En el cas del meu programa utilitza quatre taules: una per a l’última valoració que es va donar a aquell node, d’altra per a l’última millor jugada que es va calcular, l’altra per a la funció d’avaluació (cada cop que s’avalua una posició es guarda aquí i si es torna a donar retorna aquest valor) i la darrera per a portar el compte de la repetició de posicions (si una posició es repeteix tres cops en una branca es declara taules).

També cal dir que a les posicions finals, quan hi ha poques peces, el *hash* conté el 80% de les posicions investigades, el que fa que s’acceleri notablement els càlculs en aquesta part del joc.

Les taules de *hash* són una manera eficient per a calcular la repetició de jugades i determinar si la posició serà taules per aquest motiu o no, per això tots els programes les incorporen.



5. ALTRES MILLORES

Hi ha un seguit de millores que no representen avenços en la manera en què la màquina pensa, són simplement ajudes i extres que no són del tot necessaris, però si recomanables. Tot i que moltes d'aquestes millores el meu programa no les implementa són interessants d'afegir en un futur.

5.1. LLIBRE D'OBERTURES

Una millora molt efectiva és l'ús dels llibres d'obertures. En els escacs els primers moviments estan molt estudiats i cada línia conté les seves idees pròpies, els seus moviments correctes, incorrectes i arriscats... I per això al llarg dels últims 100 anys s'ha escrit molta literatura intentant explicar pausadament per als humans. En aquest aspecte es destaca la ECO (*Encyclopaedia of Chess Openings*), on des del 1966 es va actualitzant i conté totes les obertures existeixen que tinguin nom propi (per exemple: Obertura Holandesa Gambit Stauton).

Inicialment no es van aprofitar d'aquesta informació i van decidir guardar les taules de transposició dels primers moviments i utilitzar-les a totes les partides i fer-la més gran segons el temps que passés. Després van decidir agafar els llibres escrits i els anaven ficant a mà en una base de dades, de manera que el programa podia veure aquestes dades. I per últim es va decidir que els propis programes, investigant sobre la posició inicial, construïssin llibres d'obertures propis. A més, si a tot això sumem la quantitat de partides d'alt nivell que són registrades cada any, tenim una gran quantitat d'informació sobre les primeres 20-30 jugades.

Per exemple amb totes aquestes tècniques existeixen llibres com el del programa "Shredder" on el seu llibre d'1 GB conté les millors 16 milions de jugades per als 20 primers moviments.

Aquests "Llibres" actuen com una base de dades, on es pot examinar la informació, afegir-ne més o eliminar-ne les de baix nivell.

5.2. TAULES DE FINALS

Un altre manera d'aprofitar les bases de dades són les taules de finals que contenen posicions amb poques peces i la manera més òptima per a guanyar, fer taules o resistir la derrota. Aquestes taules es creen a partir d'anàlisis retrògrades. Per tal d'explicar més fàcil això de les "anàlisis retrògrades" posem per exemple la generació d'una taula que contingui totes les posicions i la manera de donar escacs i mat a cada posició del mat de dama i rei contra rei:

Primer es col·loca el rei defensor a qualsevol banda del taulell. Per simetria, només cal posar el rei en 10 caselles diferents (si col·loquem el rei en qualsevol dels quadres del triangle a1-d4-d1 es pot aconseguir les altres 64 posicions rotant el taulell). Es generen aquests deu taulells. A cada taulell hi ha 64 maneres de col·locar l'altre rei, i 64 maneres de col·locar la dama. Doncs



ALGORISMES APLICATS ALS ESCACS: PROGRAMACIÓ D'UNA APLICACIÓ

són un total de $64 \cdot 64 = 4096$ a cada un dels 10 taulells. Així generem 40960 posicions i les guardem en una taula.

Un cop es té aquesta taula s'analitza que tot sigui legal: que no hi hagi dos reis enfrontats. I totes aquestes posicions s'eliminen de la taula. Llavors es va taulell per taulell i es determina si hi ha escacs i mat o ofegat o cal investigar més. Un cop se saben totes les que són mate, s'agafa una i determina com podria haver estat a la jugada anterior, s'agafen totes les posicions "candidates" i es determina si li tocava moure al bàndol fort. D'aquesta manera recursivament si hi ha una posició que es pot donar mat en n jugades, en la posició candidata es podrà donar mate en $n+1$ jugades. Un cop s'ha acabat d'investigar-ho tot, cal repassar que la feina esta ben feta.

Aquesta idea es pot extrapolar per als altres finals i a dia d'avui existeixen taules de finals que contenen tots el finals on hi ha 7 o menys peces sobre el taulell. Per desgràcia no es factible seguir avançant, ja que les actuals, anomenades "Tablebases de Nalimov", utilitzen grans sistemes de compressió i ocupen 1.2 TB. Tot i així es continua avançant i es destrossen creences que en el món dels escacs es donaven per certes, com per exemple Dama i peó contra Dama són taules, perquè el bàndol defensor sempre trobarà escac continu, però si s'utilitza aquestes taules es guanya el 80% de les posicions.

Aquestes taules ajuden a prevenir l'efecte horitzó al final de la partida i a més ajuda a la velocitat del programa ja que alguns nodes es poden considerar terminals si hi ha menys de set peces sobre el taulell.

5.3. AUTOREGULACIÓ DEL TEMPS

En els escacs de competició és bàsic que el jugador o programa decideixi quan ha investigat prou i ha de moure o ha de seguir investigant. En aquest cas intervenen dos factors: quantitat de temps a usar i dificultat de la posició. Si la posició és "difícil" gastarem "molt" temps, si la posició és "fàcil" en gastarem poc. El que costa de definir és com n'és de "difícil" una posició i com de "fàcil" és una altra. Hi ha dos mètodes per a fer-ho en qualsevol tipus de partida, tant ràpides (5 minuts per jugador) com lentes (més d'un hora per jugador). El primer funciona així:

1. Al principi es calcula el temps mitjà que s'hauria de gastar fent una divisió, entre el temps total que li queda i el número de jugades que queden fins el següent control de temps o fins la jugada 50 (la majoria de partides s'acaben en aquestes jugades). A aquest temps mitjà l'anomenem T .
2. Llavors el programa investiga la posició amb la profundització iterativa durant $T/2$. Si només hi ha un node interessant en el primer nivell de profunditat es fa aquella jugada. Entenem per node interessant aquell que reporta un valor més alt que els altres fills. Si no hi ha aquest "node interessant" es continua investigant a més profunditat.



ALGORISMES APLICATS ALS ESCACS: PROGRAMACIÓ D'UNA APLICACIÓ

3. Un cop s'arriba al temps T es calcula la diferència que hi ha entre el resultat anterior fet en un temps T/2. Si és molt diferent es continua investigant perquè segur que hi ha problemes a l'horitzó de la cerca. Si són semblants es fa aquella jugada.
4. Es repeteix l'operació anterior per a 2T, 3T i 4T. A 4T se li obliga a fer l'última jugada que ha considerat millor.

L'altre mètode el va proposar Robert Hyatt al 1984 i es basa en la observació del joc dels grans mestres a les partides de torneig. Aquest mètode és més determinista que l'anterior ja que diu exactament quant ha de gastar abans de començar. Hyatt s'adonà que els humans gastem menys temps a les 10 primeres jugades (perquè es tracta de l'obertura i ja està estudiada) i als últims moviments, a partir de la jugada 40, tornem a gastar menys (perquè els finals són tècnica no càlcul, només fa falta conèixer-los). En els valors entremetjats arribem a gastar fins a 20 vegades més temps, tenint el màxim a la jugada vint. Els valors entre l'obertura i aquest punt més alt creix molt ràpidament i després és més progressiu. En aquest mètode per a fer-lo relatiu al temps s'ha de definir un factor que multiplicat pel temps que quedi doni el temps que s'ha d'utilitzar.

Tot i així hi ha la modalitat Fisher en els escacs en que el temps es va incrementant en cada moviment o les modalitats que es basen en controls, i per tant un sol algoritme no pot treballar bé a totes les modalitats, però aquests dos són bastant útils i genèrics.

A més la majoria de programes tenen la opció de "ponderar" un moviment mentre el rival pensa. Això vol dir que el programa intenta endevinar quina jugada farà el rival i durant el temps del rival pensarà que pot fer en el suposat cas que fes aquella jugada. Això fa que el programa aprofiti el temps del rival i així poder respondre abans la rèplica d'aquell moviment.

5.4. LOG DE LA PARTIDA, EL PGN

Per tal de guardar partides la majoria de programes utilitzen el format PGN (*Portable Game Notation*). Es tracta d'un mètode senzill que s'emmagatzema en fitxers de text amb extensió ".pgn".

El fitxers PGN comencen amb set etiquetes que descriuen extres de la partida: el nom del jugador de blanques, de negres, lloc, data, ronda, resultat i esdeveniment on es va jugar. També es poden afegir algunes etiquetes extres com l'hora o com va acabar la partida (abandonament, escac i mat...). Una etiqueta es defineix en una línia que comença amb un "[" i acaba amb un "]". Dins els claudàtors es col·loca la informació que es vol transmetre en anglès (*Event, Site, Date, Round, White, Black, Result*) i a continuació, entre cometes dobles es col·loca la informació (*World championship, Bonn, ...*).

A continuació, s'escriu la partida en notació algebraica (1. e4 e5 2. Nf3 ...) On s'agafen les inicials angleses per a les peces: Cavall N (*Knight*), Alfil B (*Bishop*), Torre R (*Rook*), Dama Q (*Queen*), Rei K (*King*). Si hi ha un enroc s'utilitza la lletra O (no el número zero), (O-O enroc



ALGORISMES APLICATS ALS ESCACS: PROGRAMACIÓ D'UNA APLICACIÓ

curt i O-O-O enroc llarg). Si hi ha escac es col·loca un signe '+' al final del moviment i si és mat un '#'. Al final de la partida es col·loca el resultat (0-1, 1-0 o $\frac{1}{2} - \frac{1}{2}$) o un asterisc si la partida no ha acabat.

A més es poden posar més d'una partida en el mateix fitxer. Per exemple aquest és el pgn de la primera partida del meu programa contra mi:

```
[Event "1st Game"]
[Site "Computer"]
[Date "2010.09.17"]
[Round "-"]
[White "VIC Is Chess"]
[Black "Black Player"]
[Result "1-0"]
1. e3 e5 2. d4 e4 3. d5 f5 4. Qd4 Nf6 5. Qe5+ Be7 6. Qxf5
d6 7. Qg5 O-O 8. a4 Ng4 9. Qh5 Nxf2 10. Be2 Nxf1 11. Bg4
Bh4+ 12. g3 Qf6 13. Bxc8 Qf2+ 14. Kd1 Qxg1+ 15. Kd2 Rf2+ 16. Kc3
Bf6+ 17. Kb3 Qxc1 18. Be6+ Kh8 19. Qe8# 1-0
```

A més en el CD adjunt podeu trobar el fitxer PGN amb totes les partides

5.5. PROTOCOLS UCI I XBOARD

De moment no ens parat a pensar sobre la interfície gràfica del programa. Generalment la GUI (*Graphical User Interface*) va a part del programa i permet gestionar fàcilment molts de mòduls d'anàlisi: permet fer-los jugar entre ells, gestiona el taulell, el temps... Per tal d'establir la comunicació entre la interfície i els motor s'ha ideat dos protocols: el *Chess Engine Communication Protocol*, conegut com Xboard o WinBoard perquè són les interfícies que el van suportar primer, i el *Universal Chess Interface*, reconegut per les sigles UCI.

La primera interfície que va aparèixer va ser Xboard de la mà de Tim Mann al voltant de 1990 amb la idea d'utilitzar-lo en GNU Chess, després per a servidors d'Internet que juguessin a escacs i a partir del 1994 es començà a popularitzar i va néixer el programa "Xboard" tal com el coneixem avui. El protocol permet jugar a molt tipus de variants dels escacs i diferents controls de temps (amb increment o sense, amb control o sense o qualsevol combinació). Després a principis del 2000 va aparèixer el protocol UCI dissenyat per Stefan Meyer-Kahlen, autor del programa "Shredder" i es va presentar com una millora de l'anterior ja que aquest protocol conté tot el que l'Xboard té i a més fa que la interfície tingui més pes que abans. Per exemple: el llibre d'obertures o les taules de finals les domina la interfície i no el programa. Aquesta aparent millora té els seus retractors que prefereixen que el mòdul d'anàlisi domini tant el llibre d'obertures com les taules de finals, perquè defensen que aquests camps defineixen el joc del mòdul i això no ho pot fer la GUI. Ambdós protocols són de codi lliure.



ALGORISMES APLICATS ALS ESCACS: PROGRAMACIÓ D'UNA APLICACIÓ

En quant a la manera d'actuar són molt semblants: és tracta d'un "diàleg" amb text formatat on interactuen la GUI i el programa en qüestió. Aquests textos són relativament senzills d'entendre ja que avisen al mòdul quan ha de pensar o ponderar i el mòdul, el que retorna són les jugades o demana coses com el temps que li queda. També cal dir que el protocol UCI és més prohibitiu en el sentit que força a programa a pensar més o menys temps o tants nodes i també li diu quant ponderar la següent jugada, mentre que el Xboard dóna més llibertats deixa que el mateix programa esculli el temps i com emprar-lo. Aquesta diferència és més notable a l'hora de ponderar.



Una de les GUI més populars és ARENA, en aquesta imatge es veu gestionant 4 motors d'anàlisi alhora

Els dos protocols són senzills de programar per això la majoria de programes implementen els dos i, fins i tot, existeix *Polyglot*, un programa que permet traduir d'un protocol a un altre sense necessitat que un mateix programa implementi els dos.

5.6. OFERIMENTS DE TAULES I RENDICIÓ

Una cosa important a decidir quan es juga una partida és en quin moment la partida està perduda i s'ha d'abandonar o quina posició és taules i s'han d'oferir o acceptar-les. Per tant, en competicions de motors d'anàlisi es deixa aquesta decisió al programa.

Primer examinarem quan s'ha d'abandonar. L'abandonament s'ha de fer quan la partida estigui objectivament perduda, per tant és millor deixar-lo el més llunyà que es pugui. Una heurística senzilla és la següent: posem la màquina a "pensar" amb el Minimax a una profunditat petita, però segura (3 o 4 nivells de profunditat). Si el valor del node pare està molt inclinat cap al rival, per exemple una torre de diferència, el programa abandonarà. També es podrien fer d'altres més complicades especulant amb el temps (si al rival li queden 10 segons, tot i que tingui la posició guanyada, no abandonarem) o amb el nivell de joc del rival (si no ha fet les últimes millors jugades, podem pensar que s'equivocarà més endavant). Tot i així la majoria de programes juguen fins a les últimes conseqüències, cosa que els humans no fem.



ALGORISMES APLICATS ALS ESCACS: PROGRAMACIÓ D'UNA APLICACIÓ

En segon lloc està el tema de les taules. Partirem de dos casos: la posició és realment taules o la posició no és del tot taules però hi ha joc igualat o hi ha un petit avantatge per a un bàndol. I a cadascun d'aquests casos examinarem si és el programa qui ha d'oferir taules i que és el que ha de fer si el rival les ofereix.

- La posició és taules: en aquests casos s'ha d'oferir taules si es troben en el final (hi ha entre 10 i 15 peces sobre el taulell) i les ha d'acceptar si no hi ha cap possibilitat d'especular amb el temps (ambdós programes tenen més d'un minut). Si en canvi es troben en mig de la partida és recomanable no oferir-les ni acceptar-les, a no ser que el temps ens sigui desfavorable.
- La posició encara no està decidida: són aquells casos que el valor Minimax del node pare està entre 0.2 i -0.2 (suposant que un peó valgui 1). Generalment en aquests casos les taules no s'accepten ni s'ofereixen, perquè aquests valors signifiquen que la partida encara pot avançar. Però si hi ha un moment en què el programa que va guanyant no té temps o en té molt poc, haurà de donar taules (tant oferir-les com acceptar-les) i el rival, en funció del que es pugui especular acceptar-les o no. També pot ser que es trobin en un final de taules i el programa no el tingui en la seva taula de finals i per tant veu un petit avantatge que a llarg termini no es realitza, per tal d'evitar això s'hauria de considerar que entre 20 o 50 jugades que tenen aquests valors són taules i oferir-les (o acceptar-les).

No cal dir que si la posició és inferior per a un programa i l'altre li ofereix taules, el primer haurà d'acceptar-les.

També cal dir que la majoria de programes ignoren literalment aquests oferiments o la possibilitat d'abandonar, fet que fa que moltes vegades el joc sigui sense sentit o que es juguin finals que els dos programes coneixen el resultat o fins i tot que es desaprofiti la possibilitat que el rival s'equivoqui i ofereixi taules.

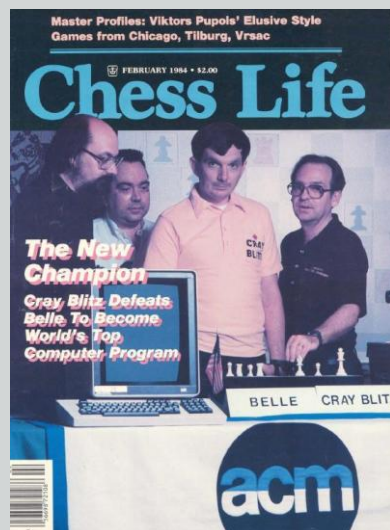
Aquestes implementacions seria interessants afegir-les al meu programa.



6. ENTREVISTA AMB ROBERT HYATT

Per tal de millorar la comprensió de tot el que s'ha explicat en aquest treball i per preguntar sobre l'evolució i futur dels escacs en el món de la computació vaig dur a terme una entrevista amb el doctor en ciències de la computació, Robert Hyatt.

Anteriorment l'he anomenat a dues ocasions: va crear el “truc” de les *mailbox* per a generar moviments i com a programador del motor d'anàlisi més fort de codi lliure, Crafty. Tot i això aquestes no són les seves úniques aportacions i al llarg de la recerca de informació m'he trobat amb aquest nom més d'un cop, com per exemple la seva victòria sobre Belle utilitzant CrayBlitz (el seu anterior programa en un superordinador) a la dècada dels 80, coronant-se campió del món dues vegades, al 1983 i 1986, o la seva introducció dels “rotated bitboards” per millorar el rendiment del generador de moviments, entre d'altres. Tot i així, la seva trajectòria comença molt abans, al 1968 ja havia creat un programa, “Blitz”. I tampoc s'ha acabat: en els últims anys, Crafty ha estat jugant partides i millorant-se contínuament gràcies a un equip de desenvolupament i a les aportacions que el codi lliure permet fer.



Portada de la revista Chess Life de febrer 1984. En portada surt Robert Hyatt (centre) després que el seu programa Cray Blitz guanyés a Belle i es coronés campió del món.

L'entrevista la trobareu íntegrament als annexes (en anglès), però aquí citaré, traduiré i comentaré les respostes més rellevants o interessants per a fer-nos una idea de la visió d'un expert en aquest camp:

P: Vostè continua fent millores en el seu programa. Vostè ha estat treballant més de 40 anys en ell, avui dia té el programa alguna cosa que no li agrada o que li agradaria canviar? En quines millores està treballant actualment?

R: Un programa d'escacs, per definició, no està mai “complet”. Un sempre pot afegir més coneixement o modificar la cerca, o ambdós, en un intent de millorar-lo. El nostre treball recent (hi ha un “equip Crafty” on dos de nostres fem la programació del motor, un altre el llibre, l'altre va provant noves idees, etc.) ha estat en l'avaluació i la cerca. Probablement en 50% en cada cosa.

INS MANOLO HUGUÉ



ALGORISMES APLICATS ALS ESCACS: PROGRAMACIÓ D'UNA APLICACIÓ

En aquest sentit veiem que és el més important en un programa: l'avaluació i la cerca. Tot i que al llarg del treball com a cerca només hem mencionat a MiniMax amb $\alpha\beta$, hi ha gran quantitat de millores en la cerca. La complicació és en quin moment s'ha d'utilitzar una o d'altre, aquest tema pot arribar a ser molt complicat i alhora decisiu en el seu nivell (programant màquines campiones del món). D'altra banda la complicació de l'avaluació com ja s'ha complicat és afegir el coneixement just i necessari, per a què no vagi el programa lent, ni faci avaluacions errònies.

P: Es possible oblidar-se del algoritme MiniMax (i dels seus derivats: $\alpha\beta$, negascout...) i dissenyar un altre algoritme? Ho has intentat?

R: Molts ho han intentat al llarg dels anys. En els programes de Go, per exemple, minimax sembla estar mort. Però pels escacs, es tracta del millor que ha estat trobat, i des que els programes vencen als humans, no hi ha cap motiu que ens forci a buscar res millor.

L'essència del minimax reflexa molt bé la idea dels escacs "Maximitzar la teva puntuació mentre minimitzes la del rival" alhora que és fàcilment computable. La pregunta anava encaminada a conèixer si ell havia fet algun "experiment". Però la resposta és ben lògica: si ja tinc una cosa que funciona, per a que necessito un altre?

P: Les dames tenen una complexitat de 10^{30} i ha estat resolta en 20 anys. Els escacs en té una complexitat de 10^{120} . Podrà ser resolt amb les tècniques actuals? Ha pensat en començar un projecte per a resoldre'ls?

R: No. Aquest número és immensament gran. Dubto que algú entengui realment com de immens és, perquè pot semblar que és curt " 10^{120} ". Si tu poguessis guardar una posició per cada àtom, no hi hauria suficients àtoms al nostre sistema solar. No crec que mai estigui resolt. És clarament resoluble des d'un punt de vista teòric, no pràctic.

Resposta amb una curiositat interessant, ens porta més a prop del perquè de la irresolubilitat dels escacs mitjançant ordinadors. La seva manera de respondre això i utilitzant els àtoms del sistema solar m'ha semblat molt curiosa.

P: Un de les teves millores pròpies són els "rotated bitboard". Podries explicar-lo breument? Com ajuda en la cerca de la posició?

R: els *bitboards* han estat al voltant sempre. El problema és agafar aquesta representació i generar els moviments legals, sense utilitzar bucles. Si tu agafes una fila del taulell, els moviments de la peça N (en aquella fila) només depèn del que conté aquella fila (estem parlant de peces que "rellisquen", en aquest cas la torre o la dama). Si tu agafes el contingut de la fila, que són 8 bits d'informació, els pots indexar en una taula de 256 bits i tindràs un *bitmap* amb els quadres on la peça pot arribar, donades les peces que bloquegen aquella fila. Però per a les columnes i diagonals els bits no són adjacents. Si fas una rotació, els fas adjacents i permetes aquest truc que funciona en les 4 direccions. Hi ha un *paper* al meu lloc web on qualsevol pot mirar aquests detalls.



ALGORISMES APLICATS ALS ESCACS: PROGRAMACIÓ D'UNA APLICACIÓ

Aquesta imaginativa alternativa per a generar els moviments no la vaig entendre el primer cop que la vaig llegir, però té molta lògica i facilita (des d'un punt de vista computacional) la generació. Els bucles són computacionalment cars, i en aquests nivells s'ha de utilitzar tot el temps en cercar més lluny, aquests tipus “trucs” ajuden molt. En un futur s'hauria d'afegir aquesta idea al meu programa.



7. CONCLUSIONS

La programació d'una aplicació que utilitzés algorismes per a jugar a escacs s'ha vist concretada en el meu petit programa VIC (*VIC Is Chess*), que tot i que no juga a un gran nivell, ha estat capaç de guanyar-me alguna partida, fins i tot amb lucidesa.

Realment l'objectiu plantejat a l'hora de començar aquest treball era molt senzill: fer un programa on juguessin dos humans. Però a l'anar avançant i aprenent més i més sobre el tema, vaig centrar-me en els algorismes perquè són més entretinguts i alhora m'han portat més maldecaps (com el *hashing* de Zobrist, vaig tenir que mirar l'escrit del mateix Zobrist per entendre'l). El programa resultant és millor del que havia plantejat des d'un principi, el fet que jugui "sol" era una cosa que no m'hauria imaginat mai que hagués pogut fer.

El principal problema que s'ha plantejat és que he après a programar des de zero, i tampoc tenia cap idea de com "pensava" qualsevol dels programes que utilitzo normalment. El problemes plantejats a la OIE i els que he fet al curs de juliol que ells mateixos van organitzar m'han servit per programar petits problemes, que a dia d'avui estan molt treballats pels programadors, com l'algoritme Dijkstra (per a trobar el camí més curt entre dos nodes) o la cerca en amplada. Però jo encara continuo aprenent i hi ha algunes coses que ara les solucionaria amb dues línies de codi, i en canvi en el programa ocupa moltes més. Aquest fet fa que, el programa que ara mateix està al CD no el vegi 100% eficient, crec que simplement tornant-lo a escriure canviant poques coses podria arribar a tenir millor rendiment.

L'objectiu principal ha estat satisfet, però el programa està molt lluny d'estar acabat, tal com diu Hyatt a l'entrevista: "un programa d'escacs, per definició, no està mai complet", i per tant es pot millorar en molts aspectes. Si algú està fullejant o mirant aquest treball i creu que pot millorar el programa el que considero més raonable per millorar és la utilització dels *bitboards*: hauria de tornar a escriure per a que suportés aquesta representació. Un cop hagi fet això haurà de reprogramar la generació de moviments i adaptar-la, ja que ara mateix és molt lenta. Un cop fet això el programa assolirà un nivell molt més alt, donat que investigarà a més profunditat, i llavors em plantejaria que dominés els protocols esmentats perquè així pugui jugar partides "de competició", encara que sigui on-line.

Haig d'agrair en especial l'actuació de Josep Ferràndiz, qui ha fet el seguiment tècnic d'aquest treball. També agrair l'actuació de la meva tutora, per acceptar el treball tot i que no es tracta de la seva especialitat, a Omer Giménez, professor que va donar el curs d'estiu i m'ha ajudat en alguns problemes puntuals, a John Corredor, qui em va ensenyar a fer el meu primer "*Hello World*" en C, i per últim al doctor en ciències de la computació Robert Hyatt, per haver col·laborat deixant fer-li una entrevista. A tot aquell que hagi ajudat a fer una línia en aquest treball, vull que sàpiguen que també els ho agraeixo.



ALGORISMES APLICATS ALS ESCACS: PROGRAMACIÓ D'UNA APLICACIÓ

8. BIBLIOGRAFIA

ARTICLES/RECERQUES

EGGER MANCILA, Jorge *Desarrollo de la maestría ajedrecística computacional utilizando recursos restringidos*. Universidad de Chile, 2003.

ALLIS, L. Victor *Searching for Solutions in Games and Artificial Intelligence*. Rijksuniversiteit Limburg to Maastricht, 1994

ZOBRIST, Albert L. *A new hashing method with application for game Playing*. University of Wisconsin, 1970.

SHANNON, Claude Elwood *Programming a Computer for Playing Chess*, Philosophical magazine, 1950.

Apunts, *Apunts d'intel·ligència Artificial*. Universitat Politècnica de Catalunya, Curs 2010-2011.

PAU A. F. *Algoritmos de juegos*. Universitat Politècnica de Catalunya, 2008.

INTERNET

EMBUENA MOLINA, Miguel Angel, *¿Cómo juega un programa al ajedrez?*
<<http://www1.freewebs.com/cheoss/comoJuegaUnProgramaDeAjedrez.html>>

<<http://www.fenach.cl/docs/memoria/node1.html>>, *Índice General de la memoria*.

<<http://chessprogramming.wikispaces.com/>>, *Chess programming wiki*.

MANN, Tim i MULLER, H. G., *Chess Engine Communication Protocol*
<<http://www.open-aurec.com/wbforum/WinBoard/engine-intf.html>>

KAHLEN, Stefan-Meyer, *UCI Protocol* <<http://wbec-ridderkerk.nl/html/UCIProtocol.html>>

<<http://es.wikipedia.org/wiki/Minimax>> *Minimax – Wikipedia, la enciclopedia libre*.

<http://en.wikipedia.org/wiki/Alpha-beta_pruning> *Alpha-beta pruning – Wikipedia, the free encyclopedia*.

<http://en.wikipedia.org/wiki/Portable_Game_Notation>, *Portable Game Notation – Wikipedia, the free encyclopedia*.

<[http://en.wikipedia.org/wiki/Board_representation_\(chess\)](http://en.wikipedia.org/wiki/Board_representation_(chess))>, *Board representation (chess) – Wikipedia, the free encyclopedia*.

<http://es.wikipedia.org/wiki/Ajedrez_por_computadora>, *Ajedrez por computadora – Wikipedia, la enciclopedia libre*.

INS MANOLO HUGUÉ



ALGORISMES APLICATS ALS ESCACS: PROGRAMACIÓ D'UNA APLICACIÓ

DIÉGUEZ, Antonio, *Cómo Funcionan* <- Amyan <- Antonio Diéguez

<<http://www.pincha.cl/amyas/comofuncionan.html>>

< http://en.wikipedia.org/wiki/Zobrist_hashing>, *Zobrist hashing* – *Wikipedia, the free encyclopedia*.

<<http://olimpiada-informatica.org/>> *OIE :: Olimpiada Informática Española 2010* (material per aprendre a programar).

FACULTAT D'INFORMÀTICA DE BARCELONA, *Teoría sobre Intel·ligència Artificial*

<<http://www.lsi.upc.edu/~bejar/ia/teoria.html>>



9. ANNEXES

A. ENTREVISTA AMB ROBERT HYATT

Thank you very much for accepting to do this interview, Dr Hyatt. The questions will be focused on how you have been working all this years, so, with Crafty or Cray Blitz.

Also there are some questions orientated to talk about the algorithms that you have been using. Your answer will be used in my research work, that talks about programming of a chess engine and about the studies that talk about generic algorithms.

The questions try to be quite concrete, but I would ask you if you can to be the most explicit as possible.

Q: Your first work was Blitz. Why did you start programming "Blitz"? How old were you? What knowledge about programming did you have? How did the computers look like? What did they have?

Professor Dr. Robert Hyatt: Blitz played its first move in October, 1968. I was a junior in the Computer Science program at the University of Southern Mississippi. The machine I used was an IBM /360 model 40, 128kb of memory, 3 7mb disk drives (yes, 3 drives, 7mb each, IBM 2311 disks). Executed around 100K instructions per second. The only language I knew at the time was FORTRAN, the base language used for everything in the CS program. I later learned PL/1 but had already started the chess program in FORTRAN.

Q: Later you started CrayBlitz, it was a supercomputer, what did it have? How was it working with it? What can you say about the pressure to have results?

A: There were several machines in this family. When Cray decided to sponsor my program in 1980, we used a Cray-1, roughly 80MIPS speed, 1M 8-byte words of memory. By the time I decided to move from supercomputers (which are very hard to schedule time on) we used a Cray T932 with 32 CPUs, 2ns clock frequency (500MIPS per CPU).

Q: Your engine, Crafty, is derived from Cray Blitz. Why did you start with Crafty and stopped with Cray Blitz?

A: Getting time on a machine that sells for somewhat North of \$30,000,000.00 is not easy. We never had enough time to test by playing real games. I decided to move to the PC, where even though we were much slower, we could play enough to thoroughly test ideas. Crafty played its first move over the December Christmas Break, 1994.

Q: You still doing improvements on your program. You have been more than 40 years working on it, nowadays has it got something that you don't like, or something that you would change? In which improvements are you working nowadays?



A: A chess program, by definition, is never "completed". One can always add more knowledge, or modify the search, or both, in an attempt to make it better. Our recent work (there is a "team Crafty" with two of us that do engine programming, one for the book, one for testing and ideas, etc.) has been on the evaluation and the search. Probably about 50-50 effort to each.

Q: Which is the best improvement you have done?

A: Probably a combination of changes to the search. Forward pruning ideas, reductions. Eliminating many search extensions. And lots of code clean-up.

Q: Your program is open-source and anyone can talk to you to help you making improvements. How many people have contributed to do it? Which are the important changes have they told you to add?

A: Every now and then someone makes a suggestion that works. I generally give them mention/credit in the comments at the top of main.c that gives the history of Crafty, from version 1.0 to present.

Q: How does the programmer chess level influence a program? Can anyone start a program or you have to have IM's or GM's helping you? Did any IM or GM contribute to do Crafty?

A: If you write your own code, you really have to have some understanding of chess concepts, if the engine is going to be strong. For example, you need to understand pawn weaknesses, king safety weaknesses, etc, or your evaluation will return wrong values and cause you to lose games. If you copy code/ideas from others, you might get by with little or no chess knowledge, because you are relying on someone else. But in 1968 there was no one to rely on, and no code to copy. Ditto until the late 90's and 2000's...

I have had at least one GM that gave me lots of advice. GM Roman Dzindzichashvili² used to play marathon matches against Crafty on ICC, sometimes 24 hours non-stop, and he would call and give me feed-back on what it was doing well and what it did poorly. It was a great help in terms of the evaluation code...

Q: 1997 Match between Kasparov and Deep Blue. Was it a turning point in chess engines? What did Deep Blue posses to be better than the World Champion? What have I got do to have a machine with those results now? Are the machines definitely better than humans in chess because Deep Blue won that match? Could Crafty beat Deep Blue today? Why?

² Nom complet: Roman Dzindzichashvili, 21è millor jugador dels Estats Units. Gran Mestre des del 1977, puntuació Elo 2550.



ALGORISMES APLICATS ALS ESCACS: PROGRAMACIÓ D'UNA APLICACIÓ

A: It probably hurt computer chess more than it helped. The general public formed the opinion that "chess is solved" once the machine dethroned Kasparov. In reality, this is far from the truth. Machines will win more than they lose against top players today, but they will still lose games. There is room for improvement.

As far as "could Crafty beat DB?" that is a question with no answer. Certainly if they had remained active, nobody would beat them today. But DB was a 1980's program (basically Belle on a chip) with an improved evaluation. New search ideas have produced significant advances in skill. Probably most programs on good hardware would have good chances against DB although it would be no pushover since it is still faster than any program today, and it would be 13 years old. But DB is gone and there is no way to resurrect it.

Q: How the engines competitions are: All the participants put their program on a supercomputer and the programs play, then you know the result, or you play what Crafty says? Is there any restrictions?

A: Most events today play on chess servers. People use whatever hardware they want, commonly multiple-cpu boxes (but rarely supercomputers). The programs are connected thru the chess server and the games are played automatically. Humans are not allowed to influence games in any way.

Q: Human competitions have their own rating system, the FIDE Elo, what about engines? Do they have a system to measure the strength of every program? How does it work? Which punctuation does Crafty has?

A: Elo works for all games. Chess programs are rated just like human players. The ratings are not easy to compare since the human chess pool rarely plays games against the computer chess pool, so the cross-pollination between the pools is minimal at best.

Q: How does Crafty spends its time to think in a Blitz game? And in a long game?

A: It behaves similar to a human. It looks at how many moves to the next time control, how much time is left, then it sets a rough "target time" to use for the next move. It can move quicker, or take longer, if certain things happen. It behaves the same whether the game is very fast or very slow.

Q: Rybka, Crafty, Shredder... Which are the differences between the top level chess machines? Have they got different algorithms or is it the machine where they run?

A: These are large software systems. There are differences in the search, differences in the evaluation, all of which add up to different levels of play in various positions... We are talking about programs with 50,000 lines of code or so, so the differences can be very significant.



ALGORISMES APLICATS ALS ESCACS: PROGRAMACIÓ D'UNA APLICACIÓ

Q: It is possible to forget MiniMax (also it derived: alpha-beta, negascout...) and design another algorithm? Have you tried it?

A: Several have tried over the years. In computer GO, for example, minimax seems to be dead. But for chess, so far nothing better has been found, and since programs are beating humans at will, there is no driving force to find something better.

Q: Checkers has a game-tree complexity about 10^{30} and it has been solved in 20 years. Chess has a game-tree complexity about 10^{120} . Will it be solved with the actual techniques? Have you thought to start a project to solve it in a long term?

A: No. That number is daunting. I doubt anyone really understands how daunting because it seems like such a short number (10^{120}). If you could store one position per atom, there are not that many atoms in our solar system. I don't think it will ever be solved. It is certainly solvable from a theoretical perspective, but not practically.

Q: How do the hash tables help to the program? It is quicker to have two: one for the pawns and another one to all the pieces or it's better to have just one?

A: That's really a complex issue. Most programs use at least two, one for search results (the real hash table that is more commonly called the transposition/refutation table) and then some sort of evaluation cache, where expensive evaluation terms are computed and stored so that they can be re-used later without the computational cost. Some programs have more than two.

Q: One of your own improvements is "rotated bitboard". Could you explain it briefly? How does it help to search in the position?

A: Bitboards have been around forever. The problem is to take a bitboard representation and generate legal moves. Without using a loop. If you take a single rank of a chess board, the moves a piece on square N (on that rank) has only depends on the contents of that rank. (we are talking about sliding pieces here, in this case either a rook or queen). If you take the contents of that rank, which is 8 bits of data, you can index into a table of size 256 and get a bitmap of the squares the piece on that square can reach, given the blocker pieces present on the rank. But for files and diagonals, the bits are not adjacent. Rotating them makes this happen and allows this lookup trick to work for all 4 directions. There is a paper on my website that anyone can look at to get the details.

Q: There are two protocols, Universal Chess Protocol (UCI) and Chess Engine Communication Protocol (Xboard or Winboard).I've read that you don't like the first one. Which advantages has Xboard over Universal Chess Protocol? Which gives more freedom to the program? Which is more used?



ALGORISMES APLICATS ALS ESCACS: PROGRAMACIÓ D'UNA APLICACIÓ

A: I believe it is the engine's responsibility to decide what to ponder, when to ponder. how long to think. When to use extra time. When to move quickly. Whether to use EGTB information or not. Which opening moves to play and how they should be selected/filtered, Etc. UCI usurps a lot of that responsibility and is a drastic departure from the xboard protocol. Since xboard works fine, I've never considered the rewrite necessary to support UCI or both, particularly when I do not believe that the GUI should be involved in playing the game at all, the GUI should be a "Graphical User _Interface_", not a part of the chess player itself.

Thank you again for your time and for answering all the questions. They will be real useful for my research work. I will follow your next advances.

B. CONTINGUT DEL CD

El CD adjunt conté tres carpetes:

- ❖ La carpeta "VIC" conté el motor d'anàlisi que he programat, amb el codi font (main.cc), els executables de Linux (main) i Windows (main.exe), a més un document anomenat log.pgn amb diverses partides que he jugat contra el meu programa.
- ❖ La carpeta "Jocs" conté els jocs "Connect4" i "Tic-Tac-Toe", que vaig fer per a aprendre a programar (amb els seus respectius executables de Linux i Windows).
- ❖ Carpeta "Altres programes" conté altres codis fonts que he utilitzat per entendre els programes:
 - Crafty-23.2
 - MSCP-1.4 (*Marcel's Simple Chess Program*)
 - TSCP-1.81 (*Tom Simple Chess Program*)
 - Viper
 - DeepXadrecko-v10.
- ❖ La carpeta "Treball" conté el present document en format PDF.