Schackspelande datorer
Hur fungerar de?

Joakim Östlund
Patrik Lindegrén

19 oktober 2004

1 Sammanfattning

Schack är ett komplicerat brädspel som kräver mycket list och skicklighet för att spela. Kan man då göra en *Artificiell Intelligens* eller *AI* som kan mäta sig med en människa? För inte allt för länge sen var det en omöjlighet, men med dagens datorer och olika algoritmer går det. En schack-AI använder sig oftast av en algoritm som heter MinMax. Genom att gå igenom alla möjliga drag ett antal drag framåt och utvärdera dem efter ett visst system kan AIn välja det bästa draget. Med olika optimeringar av algoritmen och smarta förbättringar kan man få en AI som kan tävla med de bästa schackspelarna i världen.

Detta har bevisats, då schack-AIn Deep Blue 1997 besegrade den regerande världsmästaren Garry Kasparov i en bäst av 6 match. Efterföljare till Deep Blue visar även de att schack-AI klarar av att tävla med de bästa, då de gång på gång lyckas spela lika mot högrankade schackmästare och världsmästare. Dessa spelare, som för ett tiotal år sedan skulle ha fnyst åt tanken att en dator skulle kunna spela på världsnivå, har nu börjat erkänna att det en dag i en inte allt för avlägsen framtid kommer att finnas en AI som är omöjlig att besegra, ens för de bästa i världen.

Innehåll

1 Sammanfattning **2**

2 Introduktion **4**
2.1 Bakgrund 4
2.2 Syfte 4
2.3 Avgränsning 4

3 MinMax Algoritmen **5**
3.1 Beslutsträdet 5
3.2 Evaluering 6
3.3 Algoritmen 6
3.4 Alpha-Beta Pruning 6
3.5 Heuristiska Optimeringar 7
3.5.1 Datastrukturer 8

4 Schackspelande Datorer **9**
4.1 Historiska Schackdatorer 10
4.2 Dagens Schackdatorer 11
4.3 Framtidens Schackdatorer 12

5 Slutsats **13**

A Schackregler **15**

B Pseudo-kod för MinMax-Algoritm **18**

C Pseudo-kod för alpha-beta pruning **19**

2 Introduktion

2.1 Bakgrund

För att skapa en AI som spelar schack måste man lösa många problem. Schack är ett spel för två spelare som turas om att göra sina drag. Det en schack-AI ska göra är att utifrån den givna ställningen på ett schackbräde göra ett drag. För att göra ett så bra drag som möjligt finns det en stor mängd saker som AIn måste ta med i beräkningarna. AIn ska inte bara göra drag som verkar vettiga för stunden utan måste ”tänka” framåt. Att vinna en pjäs kan verka som ett bra drag, men om detta bidrar till att motståndaren får en chans att sätta matt på två drag, är det dock inte så bra.

För att en schack-AI ska kunna spela så bra som möjligt skulle den behöva beräkna alla möjliga drag som går att göra i ett helt parti, gå igenom dem och bedöma vilka drag som skulle leda till vinst. I snitt finns det 35 möjliga drag att göra vid varje spelares tur. För ett schackparti som pågår i 50 drag (25 från varje spelare) betyder detta 35^{50}, eller ca $1.5 \cdot 10^{77}$ drag. Detta är helt omöjligt att göra.

För att man ska kunna göra en AI på det här sättet måste man göra effektiviseringar. Man måste minska ner antalet drag som AIn ska analysera, genom att bara beräkna ett antal drag framåt och tidigt avgöra vilka drag som är dåliga och inte behöver undersökas vidare. Eftersom AIn bara kommer se ett antal drag framåt måste den ha en metod för att avgöra vad som är en bra ställning, så den kan jämföra dem med varandra och välja hur den ska spela. En sådan funktion måste vara väldigt kraftfull och ta med många faktorer i beräkningen.

I denna rapport kommer två termer att användas: *schack-AI*, som refererar till den algoritm som används för att simulera en mänsklig schackspelare, och *schackdator*, som avser en hårdvarumaskin dedikerad att köra en schack-AI.

2.2 Syfte

Syftet med denna rapport är att ge en grundläggande insikt i hur en schack-AI fungerar, hur man kan skapa en optimal sådan, samt de problem som kan uppstå. En historisk överblick kommer även att göras. Rapporten riktar sig i huvudsak till människor som är intresserade av att förstå hur en schack-AI fungerar, men inte vill läsa tunga tekniska beskrivningar.

2.3 Avgränsning

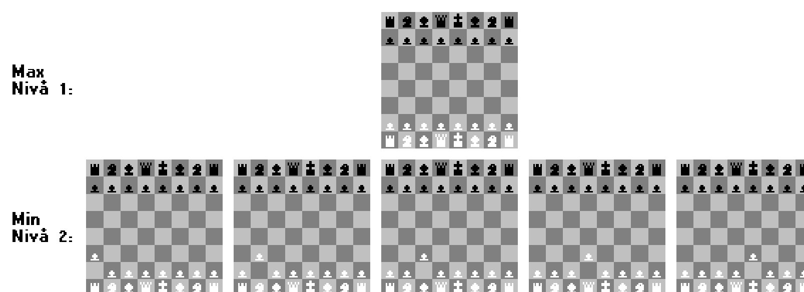
Rapporten kommer att gå igenom både historik och implementation. Av detta skäl kommer inga djupare diskussioner föras i något av ämnena, utan en genomgående överblick görs, varefter läsaren har en grund till att söka djupare kunskaper om detta önskas.

3 MinMax Algoritmen

Den algoritm som oftast används för schack-AI är den sk. *MinMax-algoritmen* [7] [8]. Denna algoritm kan användas för andra spel där två spelare turas om att göra sina drag, t.ex. *luffarschack* (även kallat ”tre i rad” eller ”tic tac toe” på engelska). Algoritmen fungerar i teorin så att den beräknar alla möjliga drag och avgör vilka drag som kommer leda till bästa slutresultatet. Detta fungerar för t.ex. luffarschack, där antalet möjliga drag inte är speciellt många och spelet endast kan pågå i nio omgångar (finns bara nio rutor på brädet). För Schack är detta dock omöjligt, algoritmen kan bara beräkna ett antal drag framåt.

3.1 Beslutsträdet

Grunden i MinMax är ett *träd* [2] [1], vilket innehåller (alla) möjliga drag (eller beslut) som går att göra i spelet utifrån ett visst läge, ett visst antal drag framåt (se Fig. 1). Trädet är därför ett *beslutsträd* och det kallas *MinMax-trädet*. Trädet består av *noder* där varje nod utgör en ställning. Algoritmen utgår från en av de två spelarna (vilken spelar ingen roll), denna spelaren kallas *Max* och den andra spelaren kallas *Min*. Vilken av dessa spelare som är AIn eller AIns motspelare och vilken av dessa spelare som börjar spelet saknar betydelse. Varje *nivå* i trädet representerar en av spelarna, och då man turas om med att göra sina drag kommer varannan nivå att representera Max drag, och varannan att representera Mins drag.



Figur 1: Två nivåer av ett MinMax-träd. På nivå 2 visas bara 5 av 12 möjliga drag. Min spelar som vit och startar på nivå 2.

3.2 Evaluering

En funktion som utvärderar spelställningen, *Eval-funktionen)*, måste finnas och är en väldigt stor del av algoritmen. Den returnerar olika poäng beroende på spelställningen, hur denna poängsättning görs är helt implementationsspecifikt, men funktionen skulle t.ex. kunna ge ett positivt resultat om det är en bra spelställning för Max spelaren och ett negativt om det är en dålig spelställning, vilket även kan tolkas som en bra spelställning för Min spelaren. Max spelaren vill därmed hela tiden försöka göra drag som leder till så högt poäng som möjligt och Min spelaren en så låg poäng som möjligt. Eval-funktionen talar helt enkelt om hur "bra" en ställning är. Testas flera olika drag och poängen jämförs från de olika ställningarna får man reda på vilket det "bästa" draget är.

3.3 Algoritmen

När MinMax-trädet skapas använder det Eval-funktionen (*Eval()*) och funktionerna *Min()* och *Max()*. Funktionerna Min() och Max() kör varandra rekursivt [2] hela tiden, detta för att varannat drag är Maxs och varannat Mins (se Pseudokod för MinMax-Algoritmen i Appendix B).

När AIn ska göra ett drag, utgår den ifrån spelställningen och algoritmen skapar ett träd med alla dragen, ned till den nivån den klarar av. För varje slutställning på slutnivån körs Eval-funktionen och ställningarna utvärderas och poängsätts. Poängen returneras upp till föregående nivå, där det draget med högst poäng (om det är Maxs nivå) och det draget med lägst poäng (om det är Mins nivå) väljs av algoritmen.

Det draget som på den nivån har valts är det som enligt algoritmen är det bästa. Algoritmen väljer hela tiden drag åt motspelaren (på motspelarens nivåer), de dragen som skulle vara absolut "bäst" (dvs de som skulle vara absolut värst om motspelaren gjorde). MinMax algoritmen försäkrar sig på det sättet om att göra de dragen som ger motståndaren minst chans att göra bra drag och förstöra. T.ex. kan det tyckas bra att göra ett drag där 13 av motspelarens 14 möjliga följddrag leder till att man sätter honom i matt. Dock är det inget bra drag om det 14:e draget motspelaren kan göra leder till att man själv hamnar i matt. Att göra detta skulle vara en chansning på att motspelaren missar möjligheten att sätta matt. Eftersom man vill att AIn ska spela utan att göra sådana chansningar (eller misstag) utgår den alltid från att motspelaren gör det bästa motdraget.

När bästa draget funnits på en nivå returneras poängen uppåt till föregående nivå och processen fortsätter. Detta sker tills man kommit hela vägen upp till ursprungsnivån, då AIn väljer att göra det bästa draget, turen går över till motspelaren.

3.4 Alpha-Beta Pruning

Alpha-Beta pruning är en metod som ofta används för att minska (optimera) antalet noder i ett MinMax-träd. Alpha-Beta pruning ger samma resultat som

MinMax, men den behöver inte gå igenom lika många noder och är därför mycket snabbare.

Algoritmen håller reda på två extra variabler, alpha och beta. De representerar minimipoängen som spelare Max är säker på att få och maxpoängen spelare Min är säker på att få. Från början är dessa värden minus oändligheten resp. plus oändligheten. När rekursionen går vidare närmar sig värdena varandra. När ett drag hittats där resultatet för Max (alpha) är större än resultatet för Min (beta) (eller omvänt om man befinner sig på en Min nivå), så behöver algoritmen inte söka vidare efter fler noder på den nivån den befinner sig på, och kan därför avbryta sökandet, och returnera upp det bästa resultatet den hittat.

Tanken bakom detta är att så länge Max spelar optimalt på noden vi hittade, finns det inget Min kan göra för att få ett bättre slutresultat, och därför spelar det ingen roll hur de andra noderna skulle se ut. I de flesta spel så är detta sant, och Alpha-Beta pruning går oftast att implementera med goda resultat. Har man dock ett spel där man kan uppnå flera olika nivåer av vinst, så kommer man nog vilja modifiera Alpha-Beta pruningen lite, alternativt använda en annan metod för att optimera sitt beslutsträd.

Det finns dock nackdelar med algortimen, den är beroende av i vilken ordning dragen analyseras på varje nivå, om de bra dragen kommer tidigt resulterar det i att algoritmen snabbt kan avbryta sökandet på den nivån. Kommer de bra dragen i slutet går Alpha-Beta pruningen igenom precis lika många drag som MinMax och det tar därför lika lång tid. När de möjliga dragen räknas fram bör de därför vara sorterade på ett sådant sätt att de drag som har störst chans att vara bra drag kommer först. I bästa fall kan antalet noder som behöver sökas igenom på varje nivå minskas så mycket att algoritmen klara att söka två nivåer längre ner i trädet än MinMax.

3.5 Heuristiska Optimeringar

Heuristiska optimeringar, eller *fallspecifika optimeringar* som det även kallas, innebär att man lägger till optimeringar till algoritmen som bara fungerar i vissa fall, men inte alla. De optimeringar man kan lägga till i en schack-AI kommer inte alltid att passa att implementera i en luffarschack-AI och tvärt om.

Optimeringarna går ofta ut på att minska antalet noder som behöver besökas i ett träd och att snabbt avgöra vilka drag som inte behöver undersökas noggrannare.

3.5.1 Datastrukturer

En stor del i att göra en bra schack-AI är att optimera algoritmer, hitta små lösningar för att göra allting snabbare. Har man en snabbare algoritm hinner man gå igenom flera drag och ett djupare träd. Det är viktigt att skriva datastrukturerna [9] på rätt sätt. I varje nod i trädet kommer hela ställningen vara sparad i minnet. Eftersom trädet kommer bestå av en stor mängd noder är det viktigt att strukturen för att spara ställningen är så liten som möjligt.

Från första början skulle någon kanske få för sig att representera ställningen som en array med 64 (8x8) heltal, där talet representerar vilken pjäs som står där. Detta räcker inte helt för att beskriva en spelställning i schack. Man måste även veta vems tur det är, om tornen eller kungen har rört på sig (de får inte ha rört på sig om det ska gå att göra en rockad) och om bönderna nyligen förflyttat sig 2 steg (se schackregler i appendix A). Om ställningen skulle representeras med t.ex. 32-bitars integer-värden skulle en nod ta $(64 + 1) \cdot 32 = 2080$ bitar eller 260 byte minne. Detta skulle vara väldigt ineffektivt och minneskrävande.

Utnyttjar man istället minnet bättre och använder en sk. bitset, kan man få ner precis samma information på ca 1/8 av minnet. En bitset är en struktur för att kunna allokera ett godtyckligt antal bitar och sedan kunna komma åt var och en för sig.

Exempel på en datastruktur för att representera ett schackbräde eller en ställning i schack.

Variabel	Antal	Storlek (Bitar)	Summa (Bitar)
Svart/Vit Pjäs	64	1	64
Typ Av Pjäs	64	3	192
Tur (Svart/Vit)	1	1	1
Har Flyttats (För Torn o Kung)	6	1	6
Summa	-	-	263

Tabellrad 1 t.ex., kan utläsas; det används 64 st 1-bitars variabler, för att hålla reda på om det som står på rutan är en svart eller vit pjäs (står det ingen pjäs där är denna variabel oanvänd). Typ av pjäs består av 3 bitar, dvs det kan vara 8 olika saker (tom ruta, bonde, bonde (En Passant), torn, springare, löpare, kung, dam).

Användning av en sådan datastruktur istället för en array av heltal skulle leda till en ordentlig hastighetsökning och minskad minnesanvändning. Allt som kan göras för att snabba upp för algoritmerna leder till att flera drag kan analyseras och AIn kan spela bättre.

4 Schackspelande Datorer

Idén om maskiner som spelar schack lika bra som, eller till och med bättre än, människor är äldre än man skulle kunna tro. Redan på 1700-talet började det dyka upp människor som påstod sig ha skapat maskiner som kunde spela schack. Denna önskan att skapa en artificiell motståndare har vuxit fram av två huvudsakliga skäl.

Det första, och kanske främsta skälet är möjligheten att kunna spela ett parti schack oavsett vart man befinner sig eller tidpunkt. En människa som bor på en mindre ort kanske helt saknar motståndare, eller så kan han vara för överlägsen. Kanske finns det andra skäl till att man inte kan eller inte vill hitta andra människor att spela med.

Det andra skälet som brukar citeras som anledning till att så mycket tid har lagts ner på att skapa en perfekt schackspelande maskin, är att schack, som problem, är lagom lätt att definiera, men svårt nog att vara en utmaning, vilket gör det till det perfekta testet för en maskin som kan analysera och tänka på ett människoliknande sätt.

Nog kan man även anta att människans nyfikenhet har drivit på utvecklingen ytterligare. För hur ser en helt perfekt schackmatch ut egentligen? Kommer vit alltid att vinna? Eller är det så att utan misstag från någon sida så kommer slutresultat alltid att bli att ingen sida vinner?

4.1 Historiska Schackdatorer

Första gången en maskin som spelar schack nämns är 1769. Wolfgang de Kempelen presenterade sin maskin, vilken han kallade ”The Turk” [3]. Maskinen blev snabbt berömd, och de Kempelen turnerade runt i Europa under många år. Efter hans död fortsatte maskinen att byta ägare, tills den slutligen hamnade på muséum i Philadelphia, där den också förstördes i en brand 1854. Några år senare avslöjades hemligheten med maskinen: den hade styrts av en människa som gömde sig inuti konstruktionen. 1868 var nästa försök, då Charles Hooper presenterade sin ”Ajeeb” maskin, vilken även den senare visade sig ha styrts av en människa.

Det dröjde dock inte länge innan de första riktiga schackspelande maskinerna dök upp. Så tidigt som 1890 konstruerade den spanska ingenjören och matematikern Leonardo Torres y Quevedo en analog dator som klarade av att beräkna Kung och Torn mot Kung slutspel. Efter det gick utvecklingen relativt långsamt fram till 1948, då matematikern Norbert Wiener presenterade sin bok ”Cybernetics”, som bland annat beskrev hur man med hjälp av en MinMax algoritm samt en evalueringsfunktion skulle kunna konstruera ett schackspelande program. 1950 kom den första varianten. John von Neumann utvecklade då ett program som var kapabelt att spela schack på ett 6x6 stort bräde, dock utan löpare. Till detta använde han en av dåtidens kraftfullaste datorer, MANIAC I.

1951 kom det första stora genombrottet. Alan Turing utvecklar för första gången ett program som är kapabelt att spela en hel omgång schack, från början till slut. Dock saknade han datorkraften att att köra programmet, och det lämnade aldrig pappersstadiet. I stället simulerade han en omgång där han själv agerade processor, och spelade mot en kollega. Den första matchen mellan människa och maskin slutade med vinst för människorna.

Datorerna blev nu allt kraftfullare, och snart började man kunna omsätta schack-AI teorierna till verkliga program. 1952 skrev Dietrich Prinz ett program avsett att lösa specifika schackproblem, och i.o.m. att John McCarthy 1956 uppfann Alpha-Beta algorithmen var nu schackspelande datorer en realitet. Redan innan två år, 1958, fanns det två program kapabla att spela ett helt parti schack. 1967 anordnades den första matchen mellan två AI spelare. På ena sidan fanns ett schackprogram skapat av de amerikanska matematikerna McCarthy och Kotok, och på andra sidan ett skapat av Moskvas Institut för Teoretisk och Experimentell Fysik (ITEP).

Under hela 70-talet ökar intresset för schackdatorer, och flera olika tävlingar och klubbar för schackspelande program och datorer uppkommer. De är dock fortfarande inte särskilt bra, även om de kan vinna över en oerfaren spelare så har de stora mästarna inga problem alls med att överlista programmen. Det är först 1980, i och med programmet ”Chess 4.6”, som en schackdator når framgångar i en större schackturnering. Samma år inför Carnegie Mellon University sitt ”Fredkin Prize”: $100.000 till den första datorn som besegrar en världsmästare i schack. 1982 vinner schackmaskinen ”Belle” en US Masters titel i schack, 1987 lyckas programmet ”HiTech” [4] som första artificiella schackspelare någonsin vinna över en stormästare (den högsta titeln inom schack).

4.2 Dagens Schackdatorer

1988 var en milstolpe inom artificiellt schackspel. Den dittills kraftigaste schackdatorn någonsin konstruerades. Den hette ”Deep Thought” (efter datorn i Douglas Adams bok ”Liftarens guide till galaxen”, som räknade ut det ultimata svaret på Livet, Universum och Allting), och var den första schackdatorn någonsin som besegrat en stormästare i en turnering (”HiTech” hade lyckats besegra en stormästare året innan, det var dock endast en uppvisningsmatch). Datorn spelade så bra att den kom på delad förstaplats med stormästaren Mikhail Tal i turneringen, trots att det var flera stormästare och även en före detta världsmästare med bland de tävlande. Tack vare sin prestation i turneringen fick Deep Thought en USCF ranking på 2745, vilket kan jämföras med världsmästaren Garry Kasparov som har en ranking på 2813 (det är dock viktigt att notera att rankingen avgörs som en skillnad mellan nuvarande ranking och ranken på dem man besegrar. Då Deep Thought aldrig spelat rankade matcher innan fick den mycket stora poängsummor per vinst).

1989 spelas för första gången en schackmatch mellan en världsmästare och en kraftfull dator. Det är Deep Thought som möter dåvarande världsmästaren Garry Kasparov. Kasparov vann utan att förlora eller spela oavgjort i ett enda parti.

1996 hände något oväntat. IBM utmanade Garry Kasparov att spela en match om 6 partier mot deras nya specialkonstruerade schackdator, ”Deep Blue” [5]. I det första partiet hände det som ingen trott var möjligt, Deep Blue besegrade den regerande världsmästaren! Kasparov återhämtade sig dock snabbt, och vann 3 och spelade lika på 2 av de resterande partierna, vilket gav honom vinsten i matchen. IBM gav dock inte upp så lätt, utan spenderade ett år tillsammans med schackstormästare åt att förbättra Deep Blue, och 1997 gick Kasparov med på en ytterligare match [6]. Kasparov förlorade dock matchen efter att ha vunnit det första partiet, förlorat det andra, spelat lika tre partier för att sedan förlora det sista partiet. De två ingenjörerna på IBM som låg bakom Deep Blue fick därmed dela på Friedman priset på $100.000 till den första artificiella schackspelaren som lyckats besegra en regerande världsmästare.

Kasparov erkände dock aldrig Deep Blues vinst, utan hävdade att en mänsklig spelare hade hjälp superdatorn att spela vissa långsiktigt taktiska drag som schackdatorer aldrig skulle ha spelat, spekulationer som underblåstes ytterligare av att IBM hemligstämplade hela projektet, och skrotade det efter vinstmatchen mot Kasparov, utan att gå med på en 3:e och avgörande match. Vidare hade IBMs lag ett stort övertag: medan de hade 100-tals publika matcher spelade av Kasparov (inklusive matcher spelade mot datormoståndare som Deep Thought samt den äldre versionen av Deep Blue), så hade Kasparov aldrig någonsin spelat mot, eller sett ens ett enda parti spelat av den nya, omgjorda Deep Blue.

På slutet av 90-talet och under början av 2000-talet kom det allt fler schackdatorer, som med mindre hårdvara och resurser än vad IBM och Deep Blue hade tillgång till, lyckades spela mycket övertygande matcher mot stora schackspelare. ”Deep Fritz” spelar lika i en match mot nuvarande världsmästaren Vladimir Kramnik, och Kasparov spelar lika mot ”Deep Junior”.

4.3 Framtidens Schackdatorer

Efter millenieskiftet så har det inte hänt så mycket inom utvecklingen av schack-AI. En av anledningarna till detta är att man har börjat nöja sig med de resultat som går att få ut från dagens superprogram. Det finns två sätt att förbättra en schack-AI, antingen genom att förbättra dess analys av en given position, så den klarar av att se vissa värdefulla taktiska situationer. Det andra är att öka dess sökförmåga, den analyserande delen. Man har dock kommit fram till att förbättringar i sök-styrkan är överlägsen förbättringar i den taktiska delen på en schack-AI, och en viss del av intresset med schack-AI har därför försvunnit. Många AI forskare har därför gått över till att fokusera på det asiatiska strategispelet ”Go” när det gäller att skapa artificiella spelprogram. Men alteftersom datorerna blir kraftfullare kommer även dagens AI att kunna besegra världsmästarna i schack regelbundet, det är bara en fråga om tid.

5 Slutsats

AIns utveckling har blomstrat under andra hälften av 1900-talet. Grundade på en relativt simpel algoritm har schackdatorerna lyckats lösa ett komplext problem: att vinna ett spel byggt på djupgående analytisk logik och taktiskt tänkande. För vissa kommer det knappast som en överraskning att datorer är bra på ett spel som enbart bygger på logik, medan andra ännu ser sig som skeptiker. Det har dock bevisats, datorerna klarar redan idag av att besegra de bästa i världen. De som arbetar inom området är överens om att det bara är en tidsfråga innan det händer konsekvent.

En schack-AI är väldigt enkel att implementera med hjälp av de enkla algoritmerna som beskrivs i rapporten. Vi har själva testat att implementera MinMax algoritmen och programmerat delar av en schack-AI. Vi har utifrån dessa praktiska tester kunnat försäkra oss om att det vi skriver om fungerar som det gör. Vi har kunnat utgå från våra resultat och göra jämförelser med andra källor. Vi är väl medvetna om vilka problem man stöter på och hur man löser många av dem. För en vanlig programmerare bör det inte vara några större problem att på detta sätt utveckla en AI som kan slå vanliga amatörer, men för att skapa en som slår de bästa i världen krävs dock lite mer.

Referenser

[1] Kimmo Eriksson & Hillevi Gavel - Diskret Matematik och Diskreta Modeller, Studentlitteratur, 2002, ISBN 91-44-02465-7

[2] Mark Allen Weiss - Data Structures and Algorithm Analysis in C, Addison-Wesley, 1997, ISBN 0-201-49840-5

[3] Tom Standage - The Turk: The Life and Times of the Famous Eighteenth-Century Chess-Playing Machine, Walker & Company, 2001, ISBN 0-802-71391-2

[4] Jack Copeland - Artificial Intelligence: A Philosophical Introduction, Blackwell Publishers, 1993, ISBN 0-631-18385-X

[5] Feng-Hsiung Hsu - Behind Deep Blue: Building the Computer that Defeated the World Chess Champion, Princeton University Press, 2002, ISBN 0-691-09065-3

[6] Bruce Pandolfini - Kasparov and Deep Blue: The Historic Chess Match Between Man and Machine, Fireside, 1997, ISBN 0-684-84852-X

[7] Leendert Ammeraal - C++ for Programmers, John Wiley & Sons, 2000, ISBN 0-471-60697-9

[8] (2004-10-19) MinMax algoritmen:
http://www.owlnet.rice.edu/ comp212/04-spring/lectures/36/

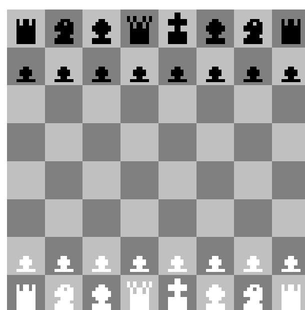
[9] (2004-10-19) Datastrukturer exempel:
http://www.freewebs.com/lasindi/chessai/

A Schackregler

Schack [1] [2] är ett spel för 2 personer där spelarna turas om att göra sina drag. Brädet, är en kvadratisk skiva indelad i 64 rutor, 8x8 st. Varannan ruta är vit respektive svart. De har 16 pjäser var, den ena spelaren har vita och den andra har svarta. Pjäserna spelarna har är 8 *bönder*, 2 *torn*, 2 *springare*, 2 *löpare*, 1 *kung* och 1 *dam*. När spelet börjar är de placerade enligt figur 2. Vit spelare börjar alltid spelet.

Målet med spelet är att sätta motspelaren i *schack matt*. Detta innebär att motspelarens kung är hotad, alla lediga rutor som kungen kan röra sig till är hotade och motspelaren inte kan göra någonting för att förhindra att kungen blir tagen nästa omgång.

I figur 3 visas hur pjäserna rör sig. Damen kan röra sig horisontalt, vertikalt

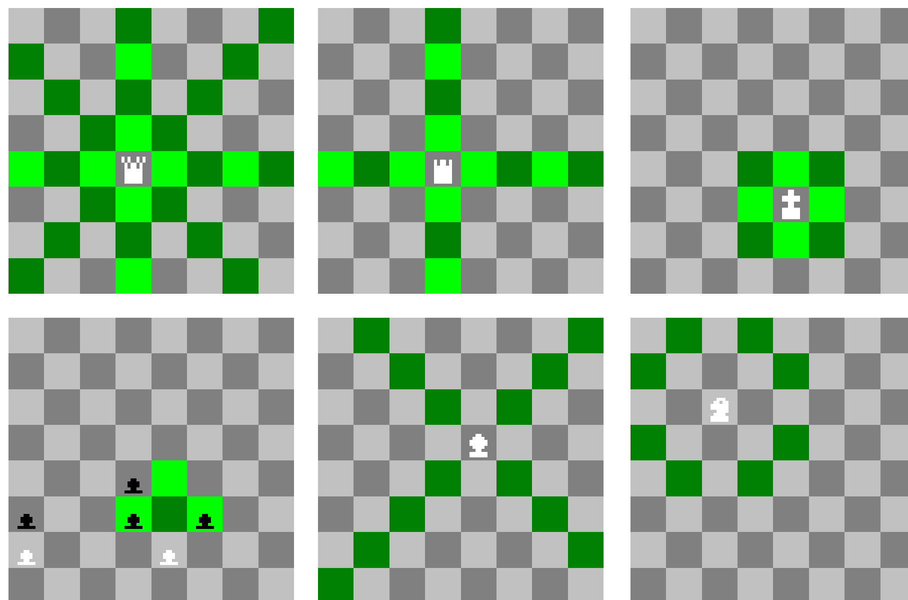


Figur 2: Schackbräde med alla pjäser utsatta i startpositionen.

och diagonalt. Tornet kan röra sig horisontalt och vertikalt. Kungen kan endast röra sig till närliggande rutor. Bonden kan vanligtvis endast röra sig 1 steg rakt framåt. Första gången den flyttas kan man välja att flytta bonden 2 steg framåt. Bonden kan endast ta pjäser som står diagonalt framför. Löparen rör sig endast diagonalt. Springaren rör sig ett steg rakt fram och ett steg diagonalt i samma riktning. Springaren är den enda pjäsen som kan hoppa över andra pjäser.

Första steget en bonde tar kan dock vara 2 steg. *En passant* regeln; om motspelaren flyttar sin bonde två steg så att den hamnar bredvid spelarens bonde, kan spelaren slå motspelarens bonde genom att gå diagonalt framåt ett steg och ställa sig bakom motspelarens bonde. Detta måste dock göras omedelbart efter att motspelaren gått ut två steg med sin bonde. Därefter är möjligheten förbi.

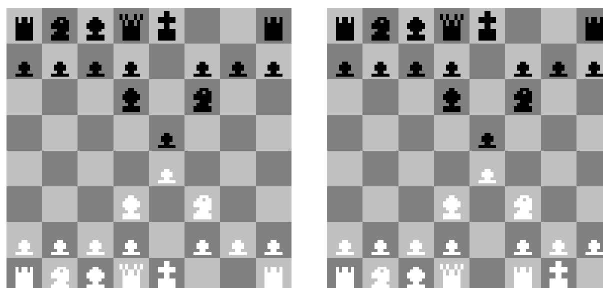
Varje omgång går till så att en spelare flyttar en pjäs till en ledig ruta, eller en ruta där en av motspelarens pjäser står, vilket resulterar i att den pjäsen är ur spel och tas av brädet. Flyttar man en pjäs så att den hotar motståndarens kung säger man att den är i schack och motspelaren måste se till att ta den hotande pjäsen, ställa någonting i vägen eller flytta kungen. Man får aldrig flytta en pjäs som sätter den egna kungen i schack. Spelarnas kungar får aldrig vara så nära varandra att de skulle kunna ta varandra. Om en spelaren inte kan göra något drag utan att sätta sin kung i schack, och inte är i schack är partiet



Figur 3: De olika pjäsernas rörelser.

oavgjort. Det är även oavgjort om ingen av spelarna längre kan vinna, t.ex. om båda spelarna bara har sin kung kvar. Om en bonde tar sig över längst ned till motståndarens sida, får denna bonde bytas ut mot valfri pjäs (dock ej en kung).

Kungen får vanligtvis endast flyttas en enda ruta åt gången. Det finns dock ett



Figur 4: Schackbärdet före och efter vit spelare gjort rockad.

undantag: *rockad* (se figur 4). Vid rockad flyttas kungen två steg, mot endera hörnet, och det torn som står där ställer sig sedan på rutan "framför" kungen (dvs den ruta som är närmare mitten). Det finns ett flertal regler som avgör om och när man kan göra rockad. Kungen får inte ha flyttats på under spelet, samma gäller för det torn som står i det hörn mot vilket man gör rockaden. Kungen får inte heller vara i schack, eller passera över en ruta där den skulle befinna sig i schack. Slutligen får det inte finnas några pjäser mellan kungen och tornet.

Referenser

[1] (2004-10-19) World Chess Federation:
http://www.fide.com/

[2] (2004-10-19) Schackregler:
http://www.chessclub.com/resources/rules/

B Pseudo-kod för MinMax-Algoritm

Min() och Max() funktionerna returnerar det "bästa" draget för spelare Min och Max. Funktionerna anropar varandra rekursivt tills de når det önskade djupet eller att spelet är slut. Eval() funktionen evaluerar ställningen och returnerar en positiv poäng om det är en bra ställning för spelare Max och en negativ poäng om det är en bra ställning för spelare Min. Koden för att skapa MinMax-trädet och få reda på "bästa" draget kan t.ex. se ut på följande sätt: Max(-1) skapar alla nivåer, Min(4) skapar trädet till nivå 4.

```
int Max( int iDepth )
{
    int iBest = -INFINITY;
    if( iDepth == 0 )
        return Eval();
    if( GameOver() )
        return Eval();
    GeneratePosibleMoves();
    while( GetNextMove() )
    {
        MakeMove();
        int iMove = Min( iDepth-1 );
        UnmakeMove();
        if( iMove > iBest )
            iBest = iMove;
    }
    return iBest;
}
int Min( int iDepth )
{
    int iBest = INFINITY;
    if( iDepth == 0 )
        return Eval();
    if( GameOver() )
        return Eval();
    GeneratePosibleMoves();
    while( GetNextMove() )
    {
        MakeMove();
        int iMove = Max( iDepth-1 );
        UnmakeMove();
        if( iMove < iBest )
            iBest = iMove;
    }
    return iBest;
}
int Eval()
{
     // Evaluerar ställningen och returnerar poäng
}
```

C Pseudo-kod för alpha-beta pruning

AlphaBeta() funktionen fungerar på ungefär samma sätt som Min() och Max() funktionerna (se Appendix B). Funktionen använder sig av samma Eval() funktion som MinMax algoritmen. Värdet för alpha ska från början vara -INFINITY och beta INFINITY. För att med AlphaBeta() funktionen göra en sökning 5 nivåer ned används följande kod: AlphaBeta(5, -INFINITY, INFINITY);

```
int AlphaBeta( int iDepth, int iAlpha, int iBeta )
{
    if ( iDepth == 0 )
        return Eval();
    if( GameOver() )
        return Eval();
    GeneratePosibleMoves();
    while ( GetNextMove() )
    {
        MakeMove();
        int iMove = -AlphaBeta( iDepth-1, -iBeta, -iAlpha );
        UnmakeMove();
        if ( iMove >= iBeta)
            return iBeta;
        if ( iMove > iAlpha)
            iAlpha = iMove;
    }
    return iAlpha;
}
```