Schackprogrammet Amundsen

(2D1464 Större avancerad individuell kurs i datalogi)

John Bergbom (d99-jbe@nada.kth.se)

27 april 2004

Innehåll

1 Bakgrund **2**

2 Ett schackpartis beståndsdelar **3**

3 Schackprogrammering **4**
3.1 Minmax 5
3.2 Alfabeta-beskärning 5
3.3 Dragsortering 6
3.4 Dödardrag 6
3.5 Aspirationssökning 6
3.6 Iterativ fördjupning 7
3.7 Transpositionstabeller 7
3.8 Kombinerade effekter 7
3.9 Nolldrag 8
3.10 Quiescence 8
3.11 Förlängning 9

4 Amundsen **9**

1 Bakgrund

Under ca 50 år har forskning bedrivits med avsikt att skapa datorprogram som spelar schack. En av de första artiklarna inom området skrevs 1950 av Shannon [6], som skisserar algoritmer för hur schackspelande ska gå till. Han beskrev två olika strategier:

- Titta på alla olika drag i en given position, och expandera alla möjliga dragsekvenser rekursivt, ända till en viss sökhorisont, dvs. ett visst antal drag framåt jämfört med nuvarande position. Denna teknik brukar kallas *brute force*.

- Gör en analys av en given position, för att bestämma de bästa dragen, och bygg sedan ett sökträd rekursivt, där endast de "bästa dragen" (och deras bästa svarsdrag) gås igenom. Detta kallas *selektiv sökning*.

Under schackprogrammeringens barndom användes i huvudsak selektiv sökning. Under hela 50-talet spelade datorerna väldigt dåligt. År 1958 presenterade Newell m.fl. [5] alfabeta-beskärning, vilken gör det möjligt att undvika att söka igenom en stor del av noderna i ett sökträd.

I början av 70-talet började de flesta program gå över till brute force, dvs. ingen selektiv sökning görs, utan alla drag beaktas i sökträdet. År 1983 blev Belle det första schackprogrammet som fick en U.S. Master-titel. Första gången ett schackprogram slog en stormästare i turneringsspel skedde 1988, och programmet hette Deep Thought.

År 1996 spelade IBM:s schackdator Deep Blue en match mot regerande världsmästaren Garri Kasparov. I denna match blev världsmästaren för första gången besegrad av en dator, i ett parti. Kasparov vann dock matchen.

Ett år senare fick IBM-teamet en returmatch mot Kasparov, och då vann Deep Blue med 3,5–2,5 i en match över sex partier. Detta är schackhistoria, eftersom det var första gången en dator slog världsmästaren i en match. Det är dock inte klarlagt att Deep Blue verkligen är en bättre schackspelare än Kasparov. Ett antal detaljer som faller utanför ramen för denna rapport, antyder att kanske Kasparov ändå är den bättre spelaren. [3] Vi kan nog säga att det inte är helt klarlagt vem som är bäst, människan eller datorn. De bästa programmen spelar mycket jämnt mot de allra bästa mänskliga spelarna.

Är det inte bara en tidsfråga innan datorerna enkelt krossar människan i schack, med ökad beräkningskapacitet? Ja, det ligger mycket i detta. Men det är samtidigt en förenkling av sanningen. Människan och datorn är bra på olika saker. Datorernas starka sida är att de är otroligt snabba, och kan bedöma 100000-tals ställningar i sekunden. En människa kan endast bedöma en eller ett par ställningar per sekund. En människa är dock otroligt överlägsen datorn när det gäller att vaska fram intressanta ställningar att bedöma. En människa går igenom en eller ett par troliga varianter, och räknar ett antal drag framåt på dessa. Detta innebär att datorn kastar bort 99% av sin tid med att bedöma värdelösa ställningar, som knappast kommer att uppkomma i praktiken. Människan är även bättre än datorn på positionell bedömning, dvs. datorn bedömer

en ställning snabbt, men människans bedömning är bättre. Datorprogram bedömer i första hand en ställning efter den materiella balansen, som innebär att man poängsätter de olika pjäserna. Dessutom ges poäng efter olika positionella faktorer såsom exempelvis ägande av en öppen linje, central placering av pjäser, osv.

Så för att besvara vår fråga: Datorerna blir bättre och bättre med ökad beräkningskapacitet, och förmodligen kommer en dag de bästa mänskliga spelarna vara helt chanslösa mot datorerna. Hur bra en dator spelar är mycket starkt beroende av hur många drag framåt den kan räkna. Den så kallade förgreningsfaktorn beskriver det genomsnittliga antalet drag som är möjliga i varje given ställning. Schack har en förgreningsfaktor på omkring 35 [1], vilket är relativt högt. Det innebär att en dubbelt så snabb dator ändå inte kommer att kunna tänka så mycket djupare. En dator måste vara många gånger snabbare än föregångaren för att kunna tänka endast ett ply[1] djupare framåt.

Det finns även vissa moment som datorn förmodligen alltid kommer att ha svårt för. En dator kan endast se det som ligger inom dess sökhorisont, vilket ger datorn stora svårigheter att hantera offer som på lång sikt kan ge en fördel. En människa kan se att ett visst offer är fördelaktigt eftersom det t.ex. ger den offrande parten ett mycket aktivt spel, vilket troligtvis ger utdelning betydligt längre fram i partiet. Viktigt att notera är att människan vet detta instinktivt, *utan att behöva räkna igenom alla kombinationer*. Detta klarar inte en dator alls. Lite förenklat kan man säga att datorn endast ser "mer material = fördel". (Datorprogram försöker även som ovan nämnts att bedöma ställningars positionella element, men de är inte på långt när lika bra som människor på detta.) Taktiska offerkombinationer vars nytta framgår *inom* datorns sökhorisont hanteras dock galant av ett program.

Ytterligare en faktor som är värd att nämna, är att det visat sig att nyttan med ökat sökdjup blir mindre och mindre ju djupare sökning man gör. [2] Det innebär att vid en viss gräns ger ytterligare sökning troligen inte så mycket bättre prestanda åt schackprogrammet. Då är det förmodligen bättre att utnyttja de extra klockcyklerna till en förbättrad evalueringsfunktion, än till att söka djupare.

2 Ett schackpartis beståndsdelar

Ett schackparti består av tre distinkta faser, som är relativt olika varandra, även om ingen väldefinierad gräns finnes mellan dem. Det är öppningen, mittspelet och slutspelet. I öppningen finns det färdiga databaser med bra drag, och utöver dessa finns enkla regler om att erövra centrum, utveckla pjäserna, ej flytta en pjäs många gånger (tempoförlust), osv. Även i slutspelet finns regler för spelföringen. Det är mittspelet som är det svåra, både för en dator och för en människa.

[1] Ply är den beteckning som normalt brukar användas inom schackprogrammering för att beteckna ett halvdrag.

3 Schackprogrammering

Det finns tre grundläggande beståndsdelar i ett schackprogram: draggenerering, sökning och en evalueringsfunktion. Sökningsmodulen skapar ett sökträd av alla möjliga kombinationer som kan uppstå ett visst antal halvdrag framåt. Draggenereringsfunktionen anropas för att skapa varje ny nod i trädet. Evalueringsfunktionen ger ett värde åt varje löv som stöts på.

I evalueringsfunktionen finns schackkunskaperna inlagda. Det absolut viktigaste attributet för att bedöma hur bra en ställning är, är att titta på hur mycket material spelarna har kvar på brädet. Hänsyn tas även till positionella faktorer, såsom dubbelbönder (ger minuspoäng), behärskande av en linje (ger pluspoäng), kungens säkerhet, osv.

Detta är grunden i alla schackprogram. Resten består till stor del av diverse trick för att snabba upp sökningen. Som beskrevs i avsnitt 1, har schack en hög förgreningsfaktor, vilket gör sökträden väldigt stora. Därför går det inte att tänka mer än 4–5 ply framåt med dagens datorer, om en fullständig genomsökning av alla dragmöjligheter ska göras, och sökningen ska gå på rimlig tid. På en Sun Ultra 1, 143 MHz, 64 Mb minne gjordes en test med en så kallad *totalsökning*, som innebär att alla drag söks igenom:

Sökning (ply)	Tidsåtgång
4	15 s
5	6 min
6	...

Tabell 1: Söktider för totalsökning.

Vid sökningen 6 ply framåt avbröt jag programmet efter två och en halv timme. Detta exempel visar att sökträdets storlek formligen exploderar för varje extra drag framåt programmet tänker. Naturligtvis går det att få ner dessa tider med någon konstant, genom att skriva snabb kod, men komplexiteten för totalsökning är fortfarande $O(b^n)$, där b är förgreningsfaktorn, och n är sökdjupet.

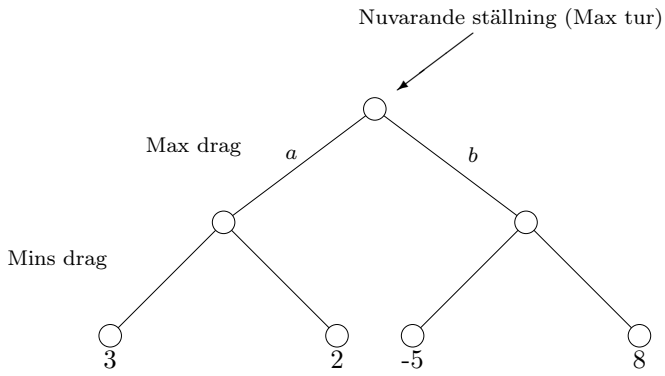
I slutspelet är förgreningsfaktorn lägre, för då finns färre pjäser kvar på spelplanen. Då går det att tänka något djupare med totalsökning (men ändå inte *djupt*).

Trots den höga förgreningsfaktorn, kan dagens schackprogram tänka över 10 ply framåt på en vanlig PC. Detta är omöjligt utan att en mängd olika trick införs som reducerar sökträdets storlek. Dessa trick beskär sökträdet så att majoriteten av alla möjliga ställningar ej behöver undersökas. Här ges en genomgång av de vanligaste tricken. För den intresserade läsaren rekommenderas webbsidan http://digilander.libero.it/gargamellachess/papers.htm, som innehåller en mängd olika forskningsrapporter över mer avancerade områden inom schackprogrammering, än vad som ryms i denna rapport.

3.1 Minmax

Vi börjar med en genomgång av den sökalgoritm alla schackprogram använder, nämligen *minmax*. För att förklara minmax kan vi tänka oss två spelare, Max och Min. Evalueringsfunktionen returnerar ett positivt värde om Max leder, och ett negativt värde om Min leder. Max ska alltså försöka göra drag som maximerar evalueringsvärdet, och Min ska försöka minimera detta värde. Bägge utgår från att den andre alltid spelar optimalt.

Låt oss anta att sökträdet i figur 1 uppkommer utifrån den aktuella ställningen. Max har två drag, a eller b att välja mellan, och Min har i sin tur två svarsdrag att göra för vart och ett av Max drag.



Figur 1: *Exempel på sökning två halvdrag framåt.*

Max vet att om han gör draget a, kommer Min att svara med draget som evalueras till 2, eftersom motståndaren alltid spelar optimalt. Om Max gör draget b gäller på samma sätt att Mins drag resulterar i ställningen som evalueras till -5. Alltså kommer Max att välja draget A eftersom det garanterar ett bra resultat, även om om han skulle få ett ännu bättre resultat om han spelade draget b, och Min gjorde ett misstag.

3.2 Alfabeta-beskärning

Alfabeta-beskärning är den mest grundläggande formen av beskärning. Trädet beskärs på ett säkert sätt, vilket innebär att antalet löv i sökträdet reduceras utan att man riskerar att missa något avgörande drag.

Det hela går ut på att man inte nödvändigtvis behöver veta det exakta värdet för varje nod i trädet. Om man har konstaterat att ett av de drag man har att välja på *kan* leda till en position med sämre poäng än vad man tidigare stött på, behöver man inte fortsätta att utvärdera vilka andra positioner draget kan leda till. Eftersom man utgår från att motståndaren alltid kommer att göra det drag som försätter en i sämsta möjliga position, vet man att detta drag kommer att leda till en position som är sämre för en själv jämfört med de drag som tidigare utvärderats.

Låt oss återgå till figur 1 en stund. Sökalgoritmen ger draget a värdet 2, dvs. det aktuella *maxdraget* (eller alfa-värdet) har värdet 2. Vid den vidare traverseringen av trädet undersöks draget b. Det första lövet som stöts på har värdet -5. Eftersom detta värde är lägre än det aktuella maxdraget, kan sökningen avslutas här. Vi vet nämligen, eftersom detta är ett *mindrag*, att b kan få ett värde av *högst* -5. Eftersom detta är sämre än maxdraget 2, kommer aldrig draget b att väljas, oberoende av vad de övriga noderna i b:s delträd har för värde.

När noder beskärs säger man att man fått en *cutoff*, och beskärningen i exemplet är en så kallad alfa cutoff. På samma sätt som det finns ett maxdrag, eller alfa-värde, finns det även ett mindrag, eller beta-värde. Detta leder till att även beta cutoffs uppstår då det omvända händer, nämligen att ett drag hittas med ett högre värde än det aktuella mindraget.

3.3 Dragsortering

Med alfabetabeskärning fördubblas det teoretiska sökdjupet jämfört med vanlig minmax-sökning (givet samma tidsmängd). Hur stor del av trädet som beskärs i praktiken är dock beroende av i vilken ordning dragen genomsöks. I exemplet ovan skulle *ingen* beskärning ha erhållits om draget som evalueras till värdet 8 hade genomsökts före det drag som evalueras till värdet -5. Idealet är om det bästa draget i varje position alltid evalueras först, för då blir beskärningen störst. Perfekt dragsortering är dock omöjlig att åstadkomma i praktiken. (Visste man redan vilket drag som var bäst, skulle man inte behöva göra någon sökning överhuvudtaget.) Med den sämsta möjliga dragordningen degenereras alfabeta till ren minmax-sökning. Därför är en viktig ingrediens till ett snabbt schackprogram att dragsorteringen är bra.

3.4 Dödardrag

Som avsnitt 3.3 beskrev, är det mycket viktigt att dragen ordnas så att de bästa dragen prövas först. Dödardrag[2] är en teknik som innebär att en förteckning görs över bra drag som orsakar cutoffs. Då en liknande position uppstår, kan drag som tidigare orsakat en cutoff testas tidigt. Förhoppningen är att draget även ger en cutoff för denna liknande position.

3.5 Aspirationssökning

I alfabetasökning sätts initialt $\alpha = -\infty$ och $\beta = \infty$. Om "fönstret" mellan maxdraget och mindraget görs smalare, kommer fler cutoffs att uppstå. Aspirationssökning innebär att en uppskattning görs av vilket värde sökträdet kommer att få, och att α/β-värdena centreras kring detta förväntade värde. Ett problem är att om uppskattningen var felaktig, så att trädets verkliga värde hamnade utanför intervallet, måste en omsökning göras.

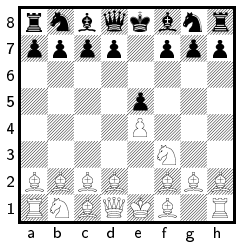
[2]Killer moves på engelska.

3.6 Iterativ fördjupning

I ett normalt schackparti har varje spelare en begränsad betänketid. Det sätt schackprogram brukar använda för att tänka en viss tid, snarare än till ett visst djup, är att iterativt tänka djupare och djupare. Det innebär att sökning initialt sker till ett fixt djup, tex. 2 ply. Sen görs en ny sökning till djup 3. Så fortsätter man ända tills antingen tiden är slut, eller det maximala tillåtna sökdjupet har uppnåtts. Leder inte detta till mycket onödigt arbete, då samma sökning görs flera gånger? Svaret är att det leder till en viss mängd extra arbete, men inte så mycket som man skulle kunna tro. Med en förgreningsfaktor x, sker sökning till djupet $(d-1)$ på endast en x:te-del av tiden det tar att söka till djupet d. I genomsnitt kräver iterativ fördjupning 4% mer arbete jämfört med att bara söka en gång. [4, avsn. iterative deepening] Fördelen är dock att tidsåtgången för schackprogrammet enkelt kan styras, och kombinerat med andra tekniker ger iterativ fördjupning faktiskt en minskning av den totala söktiden.

3.7 Transpositionstabeller

I schack finns det ofta många sätt att nå samma position. Till exempel ger öppningssekvensen 1. e2-e4 e7-e5 2. Sg1-f3 upphov till samma ställning som dragsekvensen 1.Sg1-f3 e7-e5 2. e2e4. Se figur 2.



Figur 2: *Transposition.*

En transpositionstabell är en förteckning över sökresultat för olika ställningar, som kan användas för att undvika att räkna igenom identiska positioner flera gånger. För varje nod i sökträdet som besöks, sparas värdet för ställningen undan i tabellen. Då kan denna evaluering återanvändas på andra platser i trädet då man kommer fram till samma ställning. Transpositionstabellen brukar implementeras som en hashtabell.

3.8 Kombinerade effekter

Det nämndes tidigare att iterativ fördjupning kombinerat med andra tekniker ger en minskning av den totala söktiden. Iterativ fördjupning passar mycket bra att kombinera med aspirationssökning. Det värde som erhålls vid sökning till

djup $(d-1)$ används som förväntat värde vid sökning till djup d. Då aspirationssökning används, centreras α- och β-värdena kring detta förväntade värde, så att ett högre antal cutoffs erhålls än vad som skulle ha skett utan iterativ fördjupning.

Iterativ fördjupning kan även användas för att förbättra dragsorteringen. Det går ut på att resultaten från en grundare sökning används för att sortera dragen innan en djupare sökning utförs. På så sätt ökas frekvensen cutoffs. Det bästa draget från en tidigare iteration återfinns i transpositionstabellen.

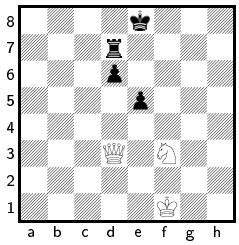
Var för sig har komponenterna aspirationssökning, iterativ fördjupning och transpositionstabell inte så stor effekt, men tillsammans leder de till en märkbar reduktion av spelträdets storlek.

3.9 Nolldrag

En teknik som ger en mycket stor minskning av sökträdets storlek är införandet av så kallade nolldrag[3]. Tekniken innebär att om man låter bli att göra ett drag, så motspelaren får göra två drag i rad, och hans ställning ändå är dålig, *vet* man att ens egen ställning är mycket fördelaktig. Om man endast söker igenom nolldraget, och nolldraget visar sig vara starkt nog för att orsaka en cutoff, har man skippat en hel ply i sökträdet. Om nulldraget inte är bra nog, får man göra en fullständig omsökning med alla legala drag istället, och i detta fall förlorar man lite tid jämfört med om ingen nolldragssökning hade gjorts alls. I genomsnitt tjänar man tid på att använda nolldragssökning.

3.10 Quiescence

Hittills har vi endast behandlat trick som snabbar upp sökningen. Nu ska vi tala lite om ett stort problem som som uppstår i samband med trädsökning.



Figur 3: *Vits tur.*

Betrakta ställningen i figur 3. Låt oss anta att sökdjupet är 3 ply. Då kommer vit att dra Sf3 x d5, i tanken att svart svarar med d6 x e5, varpå vit tar svarts torn med sin dam. Vit ser inte att i påföljande drag kommer svarts kung att ta den vita damen.

[3]Null moves på engelska.

Detta är ett exempel på det som kallas *horisonteffekten*. Ett program kan aldrig se längre än sin sökhorisont, och det kan leda till att misstag begås. Det sätt som brukar användas för att lösa detta problem, är att evaluering av löv aldrig görs på "oroliga" positioner, utan endast på "lugna" positioner[4]. Sökningen delas upp i två faser. I den första fasen genomsöks alla drag, och när man kommit till botten i denna sökning, tar den nästa fasen vid. Denna nästa fas söker endast igenom drag som slår någon annan pjäs. Eftersom antalet pjäser är ändligt, kommer denna fas att vara begränsad. Då inga pjäser kan längre kan slås, evalueras positionen. Denna metod tar bort de värsta problemen med horisonteffekten.

3.11 Förlängning

Förlängning[5] innebär att vissa speciellt viktiga eller svåra ställningar söks djupare än andra grenar. En enkel form av förlängning är att öka sökdjupet med ett varje gång en spelare hamnar i schack. Detta gör det betydligt lättare att hitta en mattsättning. Även den taktiska förmågan ökar.

4 Amundsen

Jag började skriva programmet Amundsen våren 2002. Ett fungerande program skapades, och sen låg arbetet nere under lång tid. Under sommaren 2003 tog jag upp arbetet igen, och skrev om nästan varenda rad av koden. De saker som finns beskrivna i föregående avsnitt är implementerade. Alla regler utom dragupprepning är implementerade i version 0,3.

Här kommer en versionshistorik:

- Version pre-0.1 — juli 2003
  Alfabetasökning, enda evalueringen som görs är att man tittar på den materiella balansen. Alla regler är ej implementerade. Programmet känner ej till 50-dragsregeln eller dragupprepning och dessutom tillåts rockad då man står i schack. Xboard interfacet stöds. Tidtagning stöds ej.

- Version 0.1 — augusti 2003
  Passantdrag har lagts till, samt att rockad då man står i schack ej längre är tillåten. I denna version har representationen av brädet bytts ut från en enkel array of ints till en bitboard-representation. Denna version är lite buggig. Efter ett tag börjar den göra ogiltiga drag, såsom att lämna kungen i schack.

- Version 0.15 — augusti 2003
  Dragsortering har införts. Evalueringsfunktionen tar nu även hänsyn till dubbelbönder. Snabbare kod för att hitta en bitposition i en bitboard.

[4] Quiescent positions på engelska.
[5] Extension på engelska.

- Version 0.2 — augusti 2003
  Snabbare draggenerering. Mer av arbetet görs i alfabetasökningen, för att undvika upprepning av arbete. Koden blir dock grötigare av detta. Killer moves har lagts till. I denna version förvandlar programmet alltid en bonde till en dam.

- Version 0.25 — september 2003
  Iterativ fördjupning med aspirationssökning har lagts till. Tidshantering. Förbättrad quiescence-sökning. Nu har sökmotorn valet att antingen fortsätta en slagväxling eller att acceptera värdet för positionens statiska evaluering. En test lades till för att kontrollera om motståndarens drag är giltigt. Nu hanteras 50-dragsregeln. En bugg med bitboards fixades.

- Version 0.3 — september 2003
  Förbättrad tidshantering. Stöd för increment. Stöd för att spela på FICS[6]. Nolldragsbeskärning tillagd. Bugg med rockadrättigheterna fixad. KRK- och KQK-slutspel hanteras. Check-extension är tillagd. Öppningsbok. Förbättrad evaluering. FICS rating: 1560 blitz, 1650 lightning.

[6]FICS = Free Internet Chess Server, www.freechess.org

Referenser

[1] Adriann de Groot. *Thought and Choice in Chess*. Mouton de Gruyter, second edition, 1978.

[2] Andreas Junghanns, Jonathan Schaeffer, Mark Brockington, Yngvi Bjornsson, and Tony Marsland. Diminishing returns for additional search in chess. In *Advances in Computer Chess VIII*. University of Maastricht, 1997.

[3] Garri Kasparov. My 1997 experience with deep blue. *ICCA Journal*, 21(1), 1998.

[4] François-Dominic Laramée. Chess programming. http://www.gamedev.net/reference/programming/features/chess1/, 2000. Elektroniskt dokument. Senast besökt i april 2004.

[5] A. Newell, J. C. Shaw, and H. A. Simon. Chess playing programs and the problem of complexity. *IBM Journal of Research and Developement*, 2(4), 1958.

[6] Claude E. Shannon. Programming a computer for playing chess. *Philosophical Magazine*, 41(7), 1950.