Denne guide er oprindeligt
udgivet på Eksperten.dk

COMPUTERWORLD

Introduktion til algoritmer for trækbaserede spil (Træsøgning, heuristik, minimax, alpha beta)

Her gives en kort introduktion til algoritmer (Minimax, alpha-beta afskæring, queiscense, TT), benyttet i trækbaserede spil som fx Skak, Dam, Backgammon osv. De simpleste kan implementeres i J2ME på resourcebegrænsede enheder.

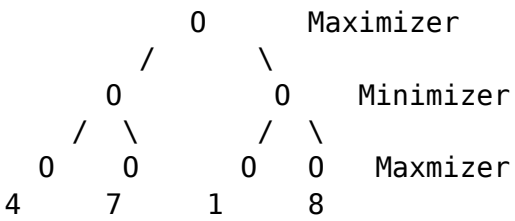
Skrevet den **10. Feb 2009** af **scheea2000** I kategorien **Programmering / Generelt** | ★★★☆☆

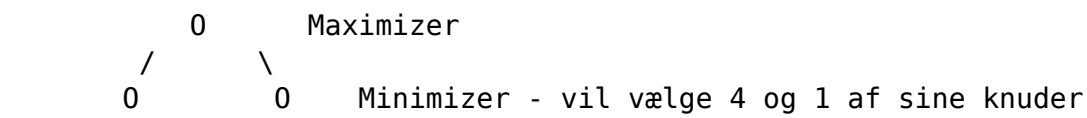
Introduktion til MiniMax algoritmen
Det er oplagt, i forbindelse med implementering af et brætspil på en computer, at indføre en form for intelligens, så programmet har mulighed for automatisk at foretage træk, og således spille mod et menneske eller en anden computer.
Dette kan bl.a. gøres ved at repræsentere hver brætsituation eller tilstand, som en knude i et træ, hvor afgreninger fra knuderne er egne og modstanderens træk.

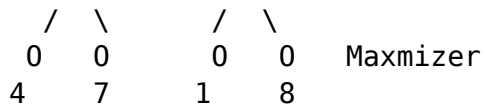
I alle tilfælde er målet med spillet selvfølgelig at vinde, men i større spil som fx skak er der langt fra computerkraft nok til at søge træet helt eller delvist til bunds og finde sluttilstanden(ene) i søgetræet (i skak ca. 10^120 knuder). Dette gør sig også gældene for simplere spil, men på resourcebegrænsede enheder som mobiltelefoner og PDA´er, som jo er relevant i denne sammenhæng.

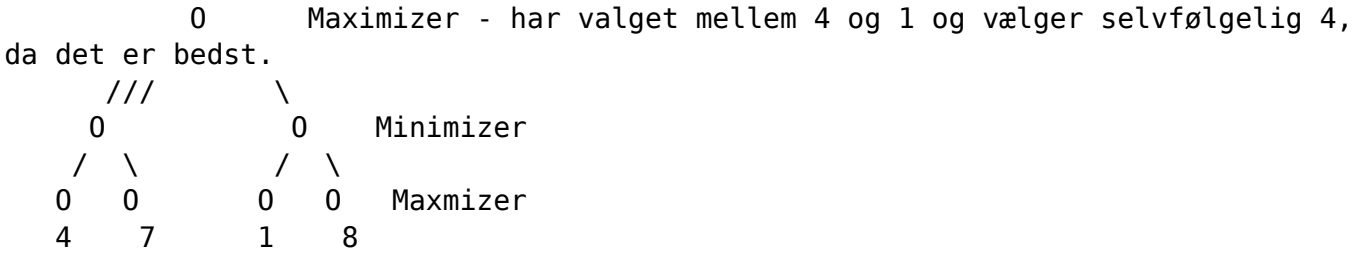
Bl.a. derfor bruges minimax-algoritmen til at gennemsøge træet. Det overordnede princip i algoritmen er, at det bedste træk vælges idet programmet forventer at modspilleren ligeledes vælger det bedste træk. Derved findes "worst-case" resultatet af modstanderens træk, som konsekvens af det træk som programmet foretager.
Knuderne bedømmer sine underknuder (børn) baseret på værdier, som findes ved at evaluere brætsituationen når den ønskede søgedybde nås. Vurderingerne foretages ud fra computerens synspunkt og det er disse vurderinger nede på "blad-niveau", som knuderne "samler" og vælger det bedste træk ud fra.
Knuderne kaldes henholdsvis for:

- Maximizere, computerens knuder, som altid vælger de største evalueringsværdier.
- Minimizere, modstanderens knuder, som altid vælger de mindste. Dermed vælges det bedste træk for modstanderen.









Det ses at den yderste højre gren er den bedste for computeren, men modstanderen (i minimizer) har mulighed for at vælge et væsentligt dårligere træk (evaluering 1) og vil højst sandsynligt gøre dette. På midterste figur ses netop den rationalisering. Minimizer-knuderne vælger de træk med mindste værdier (1 og 4).
Tilslut ender algoritmen (nederst) i den øverste maximizer, hvor det bedste træk vurderes ud fra de værdier minimizer-knuderne har returneret. Dermed har computeren valgt det bedste træk, afledt af modstanderens bedste træk.

Der er dog en lang række begrænsninger, som konsekvens af at vælge et træk baseret på denne søgningsmetode. En af dem er "Horisont effekt". F.eks. på nederste figur, hvor højre gren er valgt, baseret på evalueringsværdien 4. Konsekvensen af disse træk kunne være at spillet to træk senere for programmet er betydeligt værre (fx skatmat eller tabt!), på trods af at det umiddelbart så ud til at være bedst. Disse begrænsninger bliver beskrevet nøjere i de følgende afsnit.

Der er dog også en række forbedringer til minimax-algoritmen, både på algoritmens kompleksitet (fx antallet af beregninger, som funktion af fx søgedybde), men også spilspecifikke kvaliteter. En af dem er:

Minimax-algortimen med alpha-beta afskæring
Implementeringer af minimax-algoritmen, benytter sig næsten altid af alpha-beta afskæring, der kan ses som en optimering til den simple minimax-algoritme. Kort sagt er afskæringsprincippet, at man ikke behøver undersøge trækmuligheder for en knude, som viser sig at være bedre for spilleren på det nuværende niveau, end modstanderen vil acceptere. For at implementere denne funktion benytter algoritmen to værdier, alpha og beta, som indeholder den viden som kan kvalificere om en knude er irrelevant at søge eller ej. Princippet som følgende:

1. Første gang algoritmen køres, sættes alpha = -"uendeligt" og beta = "uendeligt".

2a. Hvis knuden er en maximizer evalueres child-knuderne. Hvis en af disses værdier er større end alpha, sættes alpha til værdien. Dette foretages indtil alle knuder er evalueret ELLER alpha < beta. Derpå returneres alpha.

2b. Hvis knuden er en minimizer evalueres ligeledes child-knuderne, men hvis en af de værdier som de returnerer, er lavere eller lig beta, sættes beta til denne værdi. Dette fortages indtil

alle child-knuder er evalueret eller alpha < beta. Derpå returneres beta.

Princippet er enkelt, men konsekvenserne er pga. Minimax-algoritmens rekursive natur, svær at overskue. Desværre har jeg ikke mulighed for at indsætte figure for at illustrere brugen og at tegne, som de andre ovenstående figure vil blive for komplekst.

Antallet af evalueringer er kritisk
Antallet af statiske evalueringer i bunden af søgetræet, er den primære "tidsrøver" i en algoritme af denne form, så hvis der ses nærmere på et træ af fast dybde d og med en fast afgreningsfaktor b, kan dette analyseres. Uden afskæring vil antallet af evalueringer være b^d.
Det kan vises , at antallet af statiske evalueringer, der skal foretages, når der anvendes alpha-beta afskæring, for et optimalt arrangeret træ, er ca. 2b^d/2-1 (d lige).
Igen må jeg beklage ligningernes form, da jeg ikke har mulighed for at gøre dem bedre overskuelighedsmæssigt.
For lethedens skyld antages det fremover at S er lig 2b^d/2 uafhængig af d´s værdi .
Et optimalt arrangeret træ, er et træ, hvor afskæring er størst og dermed giver et minimalt antal evalueringer. Det maksimale antal afskæringer er altså: b^d - 2b^d/2 og det minimale må nødvendigvis være 0.
Hvis der tages udgangspunk i eksemplet fra før, ses det at teorien giver: 2×2^2/2-1 = 3. Og det er netop antallet i eksemplet. Dermed er træet i eksemplet også et optimalt arrangeret træ. Som sagt afskæres der, hvis algoritmen har fundet en knude som giver et bedre resultatet, end modspilleren behøver acceptere, derfor må et optimalt arrangeret træ, nødvendigvis have sine bedste træk længst til venstre og dette bekræfter Patrick Henry Winston også. En metode til at tilnærme en optimal arrangering kunne være at foretage en sortering på grundlag af faktorer som f.eks. slag (captures), trusler osv.

Lad os antage at et system kan klare 1.000.000 statiske evalueringer indenfor et tidsinterval og træet er statisk med b = 10. Dvs. ligningen for algoritmen med alpha-beta afskæring ser således ud:

$$S = 2b^{d/2} \Leftrightarrow 1.000.000 = 2*10^{d/2} \Leftrightarrow \log(1.000.000) = \log(2)+\log(10)d/2$$

$$\Leftrightarrow 2* \frac{\log(1.000.000)-\log(2)}{\log(10)} = d \Leftrightarrow d =\sim 11.4$$

For algoritmen uden alpha-beta afskæring, ser ligningen derimod således ud:

$$S = 2b^{d} \Leftrightarrow 1.000.000 = 10^{d}$$

$$\Leftrightarrow \frac{\log(1.000.000)}{\log(10)} = d \Leftrightarrow d = 6.$$

Det er hermed vist at, at hvis søgetræet er optimalt arrangeret, dvs. maksimal afskæring, giver det mulighed for at søge ca. dobbelt så dybt. Hvilket giver en massiv besparelse, hvis spillet er skak, hvor træets dybde skal være maksimalt og afgreningsfaktoren er høj. Hvis det fx antages at b = 35 og der søges til dybde 6, giver det 1.838.265.625 "blade" i træet, hvor alpha-beta-afskæringen søger for (i et

optimalt arrangeret træ) at der kun evalueres 85.750 (!!).

Minimax-algoritmen er et eksponentielt problem dvs. at antallet af evalueringer stiger eksponentielt, når dybden øges eller at tiden der skal bruges på algoritmen kan udtrykkes som en eksponentiel funktion eller beskrives vha. Big-O notationen: fx O(k^n).

Elemination af horisont effekt
Hvor alpha-beta er en ren optimering af minimax-algoritmens kompleksitet - dvs. resultaterne af søgningerne er de samme med og uden alpha-beta afskæring, er eliminering af Horisont Effekt problemet en kvalitetsmæssig forbedring af spillet. Problemet består i at Minimax-algoritmen søger til et vist niveau og evaluerer. Hvis en brik lige er slået er der fx i skak stor sandsynlighed for at der kommer et modtræk, som kan falde ud positivt eller negativt, men algoritmen kan ikke "se" disse konsekvenser af dette vigtige træk pga. "horisonten". Så for at opnå et mere præcist estimat (evalueringsværdi) af forløbet videre, søges videre fx når en brik bliver slået indtil der er "stilstand" i spillet - dvs. at evalueringsværdien kun ændrer sig lidt.
Metoden hedder Search-Until-Quiescent. For at implementere en så god elimination af horisont effekten som muligt, kræver det viden omkring det specifikke spil. I skak kunne der fx være tale om "slåede" brikker, skak, brikker i slag osv.

Transposition tables
En af de mere avancerede udvidelser til den alm. Minimax algortime er Transposition tables.
Et brætspil etc. kan enten ses som et træ af mulige træk, eller som en graph hvor transpositioner kan føre tilbage til undertræer som allerede er blevet afsøgt. En tabel med transpositioner kan anvendes til at detektere disse situationer for at undgå at duplikere udført arbejde. Dette er dog langt fra den vigtigste grund til at anvende transpositionstabeller. I midtspillet vil transpositioner udgøre en meget lille del af søgetræet. Kun i specielle situationer som f.eks. et slutspil med låste bønder vil antallet af transpositioner vokse så voldsomt at brugen af en transpositionstabel gør det muligt at søge væsentligt dybere.
Den vigtigste grund til at anvende transpositionstabeller viser sig at være muligheden for at gemme det bedste fundne træk for en position. Dette giver en simpel og effektiv form for træksortering uden nogen ekstra omkostninger, som ofte vil mindske forgreningsfaktoren væsentligt.
Ifølge D. M. Breuker, J.W.H.M Uiterwijk og H.J van den Herik, anvender stort set alle skakprogrammer i dag transpositionstabeller implementeret som store hashtabeller.
Selve brætpositionen gemmes ikke i transpositionstabellen, idet dette ville kræve for meget hukommelse - i stedet gemmes den hashnøgle som repræsenterer positionen. Denne skal så være af tilstrækkelig længde til at kollisioner bliver 'meget' usandsynlige. Denne strategi stiller naturligvis også store krav til hashalgoritmen som skal distribuere værdier ensartet for at være anvendelig i praksis.
En hashtabel er generelt implementeret som et array hvor værdier indsættes på det indeks som deres hashkode, modulus længden af arrayet, foreskriver. Når to forskellige hashkoder resulterer i det samme array indeks kan forskellige strategier anvendes. De mest almindelige er chaining, hvor hvert indeks i tabellen er en hægtet liste af elementer, linear probing, hvor det første frie indeks i tabellen anvendes efter en kollision, og kvadratic probing, der ligesom linear probing placerer kolliderede elementer længere fremme i tabellen. Fælles for disse strategier er at de vil bevare alle indsatte værdier, og at hashtabellen er nød til at vokse med antallet af elementer for at en søgning efter en bestemt værdi ikke skal tage uacceptabelt lang tid. Dette er ikke ønskeligt for en transpositionstabel, eftersom træk-søgningen resulterer i millioner af mulige brætsituationer, og hvis algoritmen sættes til at søge meget dybt ville computerens hukommelse til sidst blive fyldt op, hvilket ville få programmet til at fejle.
I stedet anvendes en simpel strategi hvor der kun findes én (i nogle implementeringer dog to) værdi(er) per indeks i tabellen. Når der sker kollisioner mellem forskellige hashkoder som mapper til det samme indeks må der tages en beslutning om hvilken værdi der skal bevares.

Implementation af Minimax algoritmen med alpha-beta afskæring
For overskuelighedens skyld har jeg valgt at vise kodeeksemplerne for spillet kryds og bolle, men algoritmerne (undtagen evalueringsfunktionen) er relativt generelle og kan nemt adapteres eller udvides til andre spil også.

Det overordnede princip er:

- algoritmen startes ud med alpha = -1000 og beta = 1000 ("uendelig højt" - disse værdier skal dog justeres i forhold til ens evalueringsfunktion.)
- Da algoritmen er rekursiv, skal der tjekkes for om spillet er færdigt (tabt, vundet, uafgjort)
- Hvis maksimal dybde er nået, skal der foretages evaluering

- Hvis knuden er en maximizer:
  - så skal alle underknuder afsøges, sålænge at alpha < beta, derefter returneres alpha
  - underknudernes evaluering sættes til alpha, hvis de er STØRRE end alpha.

- Hvis knuden er en minimizer:
  - så skal alle underknunder afsøges, sålænge at alpha < beta, derefter returneres beta.
  - underknuders evaluering sættes til beta, hvis værdien er MINDRE end beta.

Det kan umiddelbart virke lidt uoverskueligt, men prøv at køre algoritmen igennem manuelt på et søgetræ. Derved kan man se hvordan teknikkerne fungerer og at de rent faktisk virker mht. afskæring. Herunder ses en implementation i Java, som er skrevet direkte i formen her, så meld endeligt tilbage, hvis der er fejl i noget af koden.

I koden benyttes klassen Board. Denne klasse har i denne simple implementation bare et integer array (9) , hvis værdi angiver om feltet på brættet er tomt, optaget af kryds eller bolle.
Den har en række metoder til at flytte en brik og fortryde det igen. Dette kan sagtens gøres "direkte" i arrayet, men det er opbygget på denne måde pga. overskuelighed. Det bliver også nærmere at udvide til mere komplekse spiltyper.
Desuden er der en lille klasse GameStatus, som indeholder lidt static status information (win, loose, draw osv). Nu bør koden være selvforklarende:

```
public int MiniMaxAlphaBeta(Board board, boolean isMaximizer, int depth, int
alpha, int beta)
{

    // Win/loose condition.
    int result = CheckForVictory();

    if (result != GameStatus.running) //game end?
    {
        if (GameStatus.maximizerWins)
            return 10;
        else if (GameStatus.minimizerWins)
            return -10;
        else //draw game
            return 0;
    }

                // if max search depth is reach, then evaluate and return
    if (depth == maxSearchDepth)
    {
        return EvaluateState(board);
    }

    //count search depth
    depth++;
```

```
    int childEvaluation;

    if (isMaximizer) // the node is a Maximizer
    {
        int[] operations = GetOperations(board, isMaximizer);
        int op;
        for (int i = 0; -1 != operations[i]; i++)
        {
            op = operations[i];
            board.movePiece(op); // simulate move

            // let opponent make countermove
            childEvaluation = MiniMaxAlphaBeta(board, !isMaximizer, depth,
alpha, beta);
            board.undoMove(op); //undo piecemove
            //if value is greater, save operation that gave the greater value
            if (childEvaluation > alpha)
            {
                if (childEvaluation >= beta)
                {
                    return childEvaluation ;
                }
                alpha = childEvaluation ;
            }
        }
        return alpha;
    }
    else // must be minimizer
    {
        int[] operations = GetOperations(board, isMaximizer);
        int op;
        for (int i = 0; -1 != operations[i]; i++)
        {
            op = operations[i];
            board.movePiece(op);  //simulate move

            // let opponent make countermove
            childEvaluation = MiniMaxAlphaBeta(board, !isMaximizer, depth,
alpha, beta);
            board.undoMove(op); //undo simulated move

            //if value is less, save operation that gave the lesser value
            if (childEvaluation < beta)
            {
                if (alpha >= childEvaluation )
                {
                    return childEvaluation ;
                }
                beta = childEvaluation ;
            }
        }
    return beta;
    }
}
```

Evalueringsfunktionen
En evalueringsfunktion bør altid udnytte information omkring det specifikke spil, dvs. den er ikke generel som fx Minimax med alpha-beta afskæring.
I dette tilfælde for kryds og bolle kunne det eksempelvis være antallet af muligheder for at få tre på stribe. I skak kunne hver brik have deres egen "værdi" og få tilføjet eller fratrukket til denne grundværdi fx afhængig af deres placering på brættet, i forhold til andre brikker eller antal af mulige træk (bevægelsesfrihed).
Det der gør en evalueringsfunktion svær i mere komplekse spil som fx skak er, at den skal statisk evaluere en dynamisk spilsituation og at en lille ændring i en briks værdi, har store konsekvenser for, hvordan algoritmen "spiller".
En oplagt valg er selvfølgelig at gøre evalueringsfunktionen bedre og mere kompleks, men da evalueringsfunktionen bliver kørt millioner af gange under en søgning, vil den kræve mere regnekraft og dermed vil algoritmen ikke kunne søge så dybt og estimationen af brætsiutation vil blive dårlige alene pga. søgedybden.

Der findes derfor to overordnede retninger. Der er dem som mener at en kvantitativ løsning er bedst og at rå regnekraft og en simpel evalueringsfunktion er vejen frem. Et godt eksempel herpå er fx Deep Blue, som benyttede råstyrke fremfor "intelligens", men en simpel evalueringsfunktion (som vist stadig er hemmelig).
Og så er der dem som mener at en kvalitativ løsning er bedre og sørger for at evalueringsfunktionen giver en så præcis estimation af brætsiutationen som muligt og trenden går i denne retning. En oplagt mulighed er selvfølgelig at lade forskellige versioner af evalueringsfunktioner spille mod hinanden, men dette er ikke ensbetydende med at vinderen, spiller bedst mod en menneskelig modstander.

En forholdsvist åbent felt - specielt i skak er neurale netværk, hvor viden omkring spillet er repræsenteret implicit. Mange af de bedre skakmotore benytter sig af en kombination af neurale netværk og traditionel træsøgning, men ingen (hvad jeg kender til) har lavet en skakmotor, som udelukkende benytter sig af neurale netværk, så udfordringen foreligger.

Det var vist alt for denne gang. I er meget velkomne til at skrive kritik/ris/ros/udpege bugs.

Artiklen er delvist baseret på et afsnit fra en rapport til et fag på Ingeniør Højskolen i København om samme emne.

Se evt. mere her:
http://www.seanet.com/~brucemo/topics/topics.htm - mere skakspecifikt.

Kommentar af nanoq d. 04. Jun 2004 | 1

Jeg er hverken kodenørd, eller en person med voldsomt kendskab til algoritmer. Alligevel fandt jeg artiklen både interessant og lærerig. Et mønstereksempel på hvordan en artikel skal skrives.

Kommentar af coldray d. 22. Jun 2004 | 2

Meget dybdegående og gribende artikel! - Et pletskud!

Kommentar af benjax d. 05. Jul 2004 | 3

Glimrende artikel, letforståelig på et godt dansk. Fint kodeeksempel. Alle points værd, specielt hvis man overvejer at sysle med brætspil på sin håndholdte.

Kommentar af blackadder d. 17. Jun 2004 | 4

God artikel. Et kompliceret emne forklaret på en letforståelig måde.
En fornøjelse at læse.

Kommentar af over-load d. 26. Jun 2005 | 5

:|!!!

Kommentar af mr-kill d. 28. Nov 2006 | 6

fin artikel