Vilhelm Lindell

August 2024

1 Introduction

I detta gymnasiearbete är målet att undersöka hur man programmerar en shackdator som uttnytjar kända tekniker för att göra den mer effektiv. Schackdatorer har funnits ände sedan 1948 då Alan Turing skapade den första riktiga Schackdatorn. Sedan dess har vi utvecklat både mycket mer högpresterande datorer som kör schackdatorerna, samt mycket mer sofistikerade algoritmer och optimiseringar på mjukvarunivå som har härstammat från den enorma mängd.

I detta gymnasiearbete är målet att undersöka hur min schackdator är programmerad och förklara de algoritmer, tekniker och optimiseringar som används för att förbättra schackdatorn.

Min schackdator som jag kommer att referera till är skrivet i programmeringsspråket Rust, men eftersom koden kan vara svårtolkad om man inte är van vid den kommer jag främst använda pseudokod när jag förklarar de algoritmer som jag har uttnytjat.

När man bygger en schackdator är den vanligaste metoden att man

2 Struktur

För att representera en schackposition på ett sätt som är optimalt för datorn att hantera används så kallade Bitboards. En Bitboard är ett 64-bitar långt binärt tal där vi låter varje bit i talet representera en ruta på schackbrädet. Ifall en bit i talet är 1 eller 0 representerar därefter ifall en vis typ av pjäs finns på den rutan eller ej. 12 Bitboards används för att representera positionen av alla pjäser på spel planen där varje bitboard representerar befintligheten av en vis typ av pjäs. I exemplet nedan visas bitboarden för de vita bönderna vid startpositionen. I koden så representerar den minst signifikanta biten, den bit vars värde är $2^0, A8 på schackbrädet med anden mest signifikanta biten representerar H1, och mellan dessa är indexeringen öka$

En modern dator har en 64 bitars CPU, vilket innebär att den kan utföra operationer på 64 bitars tal väldigt snabbt på grund av hur de elektroniska kretsarna är strukturerade i den. Av denna anledning är användet av Bitboards optimalt eftersom CPUn kan utföra instruktioner direkt på den datan som representerar brädets status.

På grund av detta kan vi uttnytja många av de instruktioner som CPUn kan utföra på 64 bitars tal för att manipulera bitboardsen och därmed också

pjäserna. Två av de mest vanliga instruktionerna som används i mitt program är så kallad logisk skift https://en.wikipedia.org/wiki/Logical$_shiftsomheltenkelförflyttarallabitarietttalåtvänst$
or.

$$a \ll n$$

flyttar alla bitar ett steg åt vänster

$$a \gg n$$

flyttar alla bitar ett steg åt hoger

Två andra viktiga operationer som inte alltid är explicit instruktioner eftersom det varierar från mellan olika instruktionssätt för CPUer, men i Rust är dessa funktioner: https://doc.rust-lang.org/std/primitive.u64.htmlmethod.leading$_zerosleading_zerossomreturn$
$indexförminstsignifikantabiten.Dettaärrelevantsenare.Densistanämnvärdaoperationenpåbitboardssomär$

3 Evaluering

4 Drag generering

5 Sökning

Minimax är ett algoritm som används för att bestämma poängen efter ett visst mängd drag för ett noll-summa spel, med bästa drag enligt en evaluerings funktion. I mitt schackprogram använder jag en variation av minimax som kallas för negamax, vilket simplifierar koden genom att uttnytja följande faktum

$min(a, b) = max(-b, -a)$

5.1 Negamax

Detta fungerar således evalueringsfunktionen returnerar ett värde som är relativt till sidan som gör draget–då större värden är bättre–vilket innebär att i negamax försöker både sidorna maximera evaluerings värde.

Algoritmen fungerar genom att gå igenom ett träd av alla möjliga positioner till ett visst djup. Vi börjar vid brädets nuvarande positionen och genererar en lista av alla lagliga drag. För varje lagligt drag skapar vi en ny nod i trädet som representerar schackbrädets position efter att draget har gjorts. Vi får ett heurestiskt värde för en av dessa barnnoder genom att anropa negamax igen från barnnoden, vilket kommer att ge oss ett heurestiskt värde för hur bra positionen är för den nuvarande spelaren.

Funktionen ger ett heurestiskt värde vid varje löv-nod som utgörs av de noder som nått det förutbestämda djupet eller som saknar lagliga drag, och noder som inte är löv-noder kommer ärva värdet värdet från det största värdet av sina barn noder. Funktionen kommer därmed rekursivt gå igenom trädet av alla drag på djupet först och varje nod kommer ärva det heurestiska värdet för det bässta draget i den nuvarande positionen. Pseudokoden för algoritmet blir följande:

```
fn nega_max(depth ) {
    if (depth == 0) return evaluate();
    max = -oo;
    for (all moves)  {
        score = -negaMax(depth - 1);
        if(score > max)
            max = score;
    }
    return max;
}
```

5.2 Alpha-beta pruning

Alpha-beta pruning är en förbättring på minimax som drastiskt kan minska antalet noder som behöver sökas. Principen utgår ifrån att vi sparar ett alfa och ett beta värde när vi söker, där alfa är det minsta poängen som den maximerande spelaren är garanterad, och beta är det största värdet som den minimerande spelaren är garanterad. Alfa får ett ursprungligt värde på -oo och beta oo. Dessa två värden är de sämsta möjliga som spelarna kan få, och när vi söker igenom trädet kommer vi uppdatera dessa. Efter vi har evaluerat värdet i en nod kollar vi ifall . Principen utgår ifrån att det bästa värdet som den maximerande spelaren kan få , a, är det sämsta värdet som den minimerande spelaren kan få, och tvärtom för b.

5.3 Horisonteffekten

Ett problem som dyker upp med vårt nuvarande sökalgoritm är en effekt som kallas för horisonteffekten. Eftersom vi har ett förutbestämt djup som vi söker till förekommer det situationer då det i lövnoden görs ett drag som har ett positivt vä. Ett exempel är ifall att det i en av lövnoderna görs ett drag där vits drottning tar en svart bonde, som i detta fallet blir positivt för vit. Problemet är

att eftersom sökningen stannar vid detta djup kollar vi inte ifall det fanns en pjäs som skyddade den bonden och som nästa drag kommer ta drottningen. Detta kan lösas genom att vi inte stannar vid en nod som är instabil, dvs det finns drag som leder till en betydlig förändring i evalueringen. Det lättaste sättet att göra detta är att vi efter vår sökningen till en specifierad djuper, söker rekursivt genom alla drag som tar en annan pjäs. Detta fungerar relativt väl eftersom de drag som vanligast ger drastiska förändringar i evalueringen. Det finns fall där drag som inte tar pjäser ger drastiska förändringar i evalueringar, men dessa ignorerar vi att söka genom i vår horisontsökning eftersom det är svårt att bestämma det utan att göra draget, vilket skulle göra sökdjupet oändligt långt.

5.4 Sortering av drag

References