Effektiv schackmotor

Vilhelm Lindell

1. Introduction

I detta gymnasiearbete är målet att undersöka hur man programmerar en schackmotor och förklara de algoritmer, tekniker och optimiseringar som används för att förbättra schackmotorn. Schackmotorer har funnits ända sedan 1770-talet men det var först 1950 då Alan Turing skapade den första datorn som kunde spela schack. Sedan dess har förbättringar inom både mjukvaran som bygger upp schackmotorn, samt enorma förbättringar inom hårdvaran, lett till att dagens schackmotorer är långt mer effektiva än de var tidigare.

Ett modernt schackprogram är uppbyggt av ett antal komponenter som möjliggör en fungerande schackmotor. Först krävs ett effektivt sätt att förvara datan som bestämmer den nuvarande positionen för schackpartiet och ytterliggare värden som är användbara att förvara under programmets körning. Med detta som grund krävs det en funktion som genererar en lista över alla drag som är strikt lagliga i positionen eller pseudolagliga. En evalueringsfunktion används för att ge positionen ett heuristiskt värde vilket är väsentligt för att programmet ska kunna avgöra vilka positioner som är bättre än andra. Huvuddelen av programmet utgörs av sökningen, vars mål är att utifrån den nuvarande positionen, rekursivt söka igenom alla positioner som kan nås från den nuvarande positionen, och göra detta till ett visst djup. Grundalgoritm Min-Max används för detta, men mer optimiserade varianter av det används oftast, men i grunden uttnytjar det drag generingen samt evalueringen för att nå de möjlig noderna (schackpositionerna) som kan nås, och bestämma vilken linje som är bäst med hjälp av evalueringsfunktionen.

Den schackmotor som jag har programmerat i samband med detta projekt är skrivet i programmeringsspråket Rust, och kommer refereras till för att klargöra hur koden som kör programmet ser ut för att på så sätt få en bättre bild över hur programmet fungerar. Eftersom att Rust-koden i sig kan vara svårtolkad om man inte är van vid hur den ser ut, kommer jag i vissa fall istället använda pseudokod i mina exempel. Jag kommer utelämna stora delar av programmet eftersom koden är aldelles för lång för att kunna förklara varenda del, och jag kommer därmed fokusera på de generella algoritmerna och metoderna som jag använder.

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. Varje bit i talet kan antingen vara 1 eller 0, och värdet på biten indikerar att en typ av pjäs som till exempel vita bönder finns på den rutan som är associerad med biten, eller om rutan är tom. I exemplet kan bitboarden för de vita bönderna vid startpositionen ses. Minst 12 bitboards krävs för att representera brädan på detta vis, en per typ av pjäs och färgen på pjäsen, men ytterliggare Bitboards för

Viktigt att notera kan vara att 64 bitars talet är utlagt så att den första biten, även känt som den minst signifikanta biten, representerar A8, medan den sista biten, även kallat den mest signifikanta biten representerar H1. I <

. 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, A8 på schackbrädet medan den mest signifikanta biten representerar H1, och mellan dessa är indexeringen ökande enligt första ranken och därefter filen.

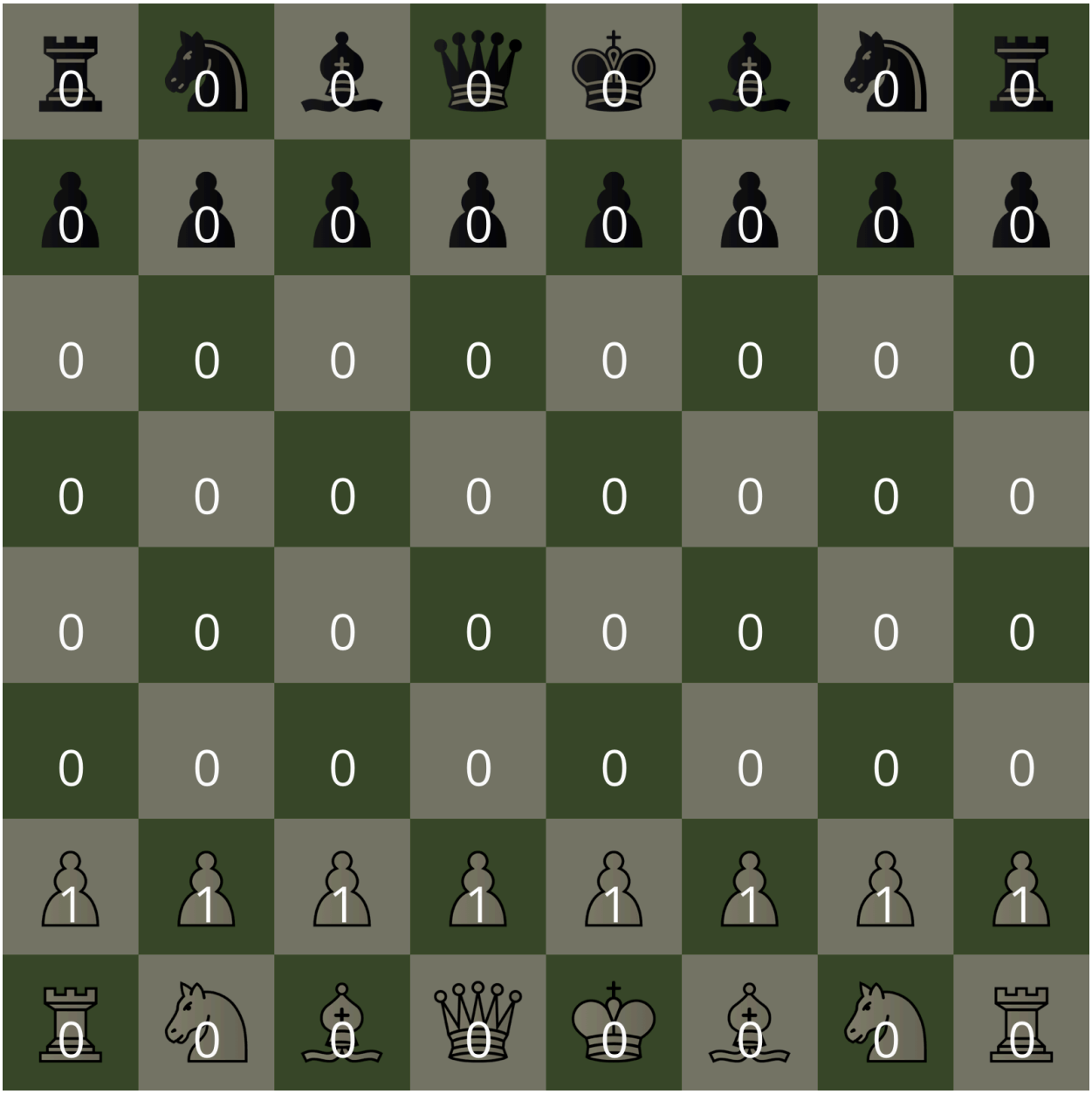

Figure 1: White Pawns

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_shift som helt enkel förflyttar alla bitar i ett tal åt vänster eller höger beroende på vilket håll skiften sker åt, och därefter fyller lediga platserna i talet fylls därefter i med 0:or.

$$a \ll n$$

flyttar alla bitar ett steg åt vänster

$$a \gg n$$

flyttar alla bitar ett steg åt hoger

$$!a$$

flippar alla bitar, varje 0:a blir 1 och varje 1:a blir 0.

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.html#method.leading_zeros leading_zeros som returnerar antalet bitar som är 0 i början av 64 bitars talet, vilket är samma som indexet för den minst signifikanta biten och trailing zeros returnerar antalet bitar i slutet av talet vilket är samma som 63 - index för minst signifikanta biten. En annan operator är bitwise not som i Rust betäcknas med ! framför variabeln eller talet.

Den sista nämnvärda operationen på bitboards som är användbar är en så kallad population count på ett 64 bitars tal som helt enkelt returnerar antalet bitar i talet som är 1, vilket för en Bitboard ger oss antalet av den pjäsen som finns på schackbrädet.

3. Drag generering

För att kunna ha draggenerering krävs en datastruktur för att representera hur ett drag ser ut. För detta krävs två tal vars värden ligger mellan 0 och 63, ett för rutan som pjäsen börjar på, samt ett för rutan som pjäsen slutar på. Utöver detta krävs ytterliggare information om vilken typ av drag det är och i mitt program representeras det med detta enum. En enum representeras internt av ett tal då talets värde korrespenderar med en av dessa alternativen som listas i enumet. På så sätt förvarar vi i draget vilken av följande typ av drag som det är. Anledningen till att dessa krävs är eftersom den typ av drag som listas här, förutom Normal och DoublePush, har någon form av sidoeffekt och att snabbt kunna kolla upp vilken typ av drag det är med en enum förvarad i draget gör att vi lättare kan kolla vilken typ av drag det är och därefter genomföra den sidoeffekt som påverkar brädet efter draget görs.

```
#[repr(u8)]
pub enum MoveType {
    Normal,
    KingsideCastle,
    QueensideCastle,
    DoublePush,
    EnPassant,
    RookPromotion,
    BishopPromotion,
    QueenPromotion,
    KnightPromotion,
}
```

Internt så kan informationen om start och slutpositionen, samt slutpositionen representeras med endast 16 bitar vilket ses i följande kod (u16 är ett 16 bitars tal), detta är möjligt eftersom typ av dragen har 9 olika värden och start och slut representeras med 0 ti

```
pub struct Move {
    bits: u16,
}
```

```
pub enum Direction {
    North = -8,
    West = -1,
    NorthWest = -9,
    NorthEast = -7,
    SouthWest = 7,
    SouthEast = 9,
    East = 1,
    South = 8,
}
```

Drag generingen är den delen av Schackmotor där användningen av bitboards blir som mest fördelaktig på grund av de operationer som effektiv kan manipulera bitboardsen. Schack har 6 olika pjäser som kan röra sig på olika sätt. Men för drag generingen är den generella principen samma, först använder vi lsb för att få positionen på brädet av en pjäs av en specifik typ. Detta låter oss loopa igenom alla 64 bitar för att hitta pjäserna eftersom vi bara bryr oss om bitarna som är 1 och därmed har en pjäs.

Därefter tar vi in en funktion som utifrån en startruta för en viss pjäs returnerar slutrutan för draget.

För att generera drag för hästen är tekniken att man vid kompileringstid genererar en array som indexeras med en ruta, ett tal mellan 0 och 64, och returnerar en Bitboard där endast de rutor som en häst som står på den rutan hade kunnat flytta sig till är 1. Detta låter oss förvarara detta värde på minnet istället för att omberäkna varje gång vi genererar drag.

Vi loopar därmed först igenom bitboarden för alla t ex vita hästar, för varje häst indexerar vi vår array och får ut alla rutor som hästen kan flytta sig till från sin nuvarande position. Vi loopar igenom dessa slut rutor och adderar ett drag till listan av varje drag.

En ytterliggare sak man måste ta hänsyn till är absoluta pins. En schackpjäs är i en absolut pin när den skyddar kungen från en attack från en glidande pjäs, och får endast flytas längs de rutorna som den glidande pjäsen anfaller. När vi genererar drag som inte är kungens drag, måste vi därmed kolla om pjäsen som vi genererar dragen för är absolut pinnad. Detta görs genom att vi håller koll på alla absolut pinnade pjäser i en bitboard, som uppdateras varje gång ett drag görs eller dras tillbaka. När vi sedan genererar drag för pjäser som inte är kungen, kollar vi först om pjäsen är pinnad, eller

```
#[inline(always)]
fn legal(board: &Board, from: usize, to: usize) -> bool {
    // a non king move is only legal if the piece isn't pinned or it's moving along the
ray
    // between the piece and the king
         board.absolute_pinned_squares.bit(from)  ==  0  ||  Board::aligned(to,  from,
board.piece_squares[Piece::new(PieceType::King, board.side)].lsb())
}
```

För att generera alla hästdrag används följande funktion

```
fn generate_knight_moves(moves: &mut ArrayVec<Move, MAX_LEGAL_MOVES>, board: &Board) {
    let bitboard = board.piece_squares[Piece::new(PieceType::Knight, board.side)];

    for from in bitboard {
        let attack_bitboard = KNIGHT_ATTACK_MASKS[from] & !board.friendly_squares();
          add_moves_from_bitboard(&|to| Move::new(from, to, MoveType::Normal), moves,
attack_bitboard, board);
```

```
    }
  }
```

den har en for loop som letar igenom alla 1:or i bitboarden för alla hästar som tillhör spelar som ska göra ett drag.

4. Evaluering

Evalueringen är tvungen att ta hänsyn till en stor mängd faktorer när den ska ta fram ett heurestiskt värde för positionen. Den enklaste metoden är att

pub fn centipawns(&self) -> i32 { match self { PieceType::Pawn => 100, PieceType::Knight => 320, PieceType::Bishop => 330, PieceType::Rook => 500, PieceType::Queen => 900, PieceType::King => 20000, } }

5. Sokning

Minimax är ett algoritm som används för att bestämma poängen efter ett visst mängd drag för ett noll-summa spel vilket är vad schack är. Minimax är beroende av en evalueringsfunktion som ger ett heurestiskt mått på hur väl det går för spelarna. 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 evalueringsvärde.

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

När vi söker igenom dragen spelar ordningen vi gör det i en stor roll. Detta beror på att alpha-beta pruningen kommer att bli mer effektiv om vi lyckas öka alfa och minska beta, dvs hitta bättre och bättre drag för de båda spelarna eftersom detta leder till fler alpha och beta cutoffs. Därmed är det viktigt att man sorterar dragen efter man har genererat dem. Det finns flera olika faktorer som kan uttnytjas för att få en effektiv sortering av drag. För att sortera dragen som i princip är en lista använder Rust något default sorteringsalgoritm där allting jag behöver skicka som argument är en funktion som tar in två drag och säger vilken som ska vara först i listan.

När man uttnytjar iterative deepening i sin sökmetod är en av de mest effektiva faktorerna för sorteringen ifall förra sökningens bästa linje innehåller det drag som man försöker sortera och vid rätt djup. Drag som man vid en tidigare sökning redan bestämt är bra har en väldigt hög chans att även vara bra vid ett djup högre, vilket är varför man vill söka dessa drag först.

En annan faktor följer tekniken MVV-LVA (Most valuable victim - Least valuable aggressor) som innebär att drag har högre chans att vara bättre dessto mer värdefull pjäsen som tas är och desto mindre värdefull pjäsen som tar den andra pjäsen är. Detta följer simpel logik då man vill ta mer värdefulla pjäser och när man tar en pjäs och om motståndarens pjäs är skyddad är det bättre att offra en mindre värdefull pjäs. https://www.chessprogramming.org/MVV-LVA Detta kan lätt implementeras i kod om man uttnytjar de förutbestämda material värdena för pjäserna genom att bara subtrahera pjäsen som blir tagens värde med pjäsen som tar den andra pjäsen. let capture_score = captured_piece.piece_type().centipawns() - moving_piece.piece_type().centipawns();

Därefter är det bra att kolla transposition table

5.5. Transposition Table

När vi söker med minimax kommer vi att stötta på samma position senare i sökträdet i två separata sökningar även om dessa sökningar börjar med olika drag. Detta beror på att samma schackposition kan nås om man bara ändrar ordningen som man göra vissa drag i. När vi söker med negamax är det därmed användbart att hålla koll på alla positioner som vi har befunnit oss i samt, spara det djup och den evaluering som gavs till positionen i tidigare sökningar. Om vi stöter på positionen igen kan vi bara återanvända evalueringen som är sparat i transpositio ntablet. I koden implementeras Transposition table med en hashtable vars hashfunktion uttnytjar den tidigarenämnda zobrist hashing. Det finns en risk att zobrist hashingen är fel eftersom det inte är en perfekt hash funktion och det finns en risk för kollisioner, men

5.6. Zobrist hashing

Zobrist hashing ‘r ett s[kallat hash algorithm .

Po’ngen med ett