NTI Gymnasiet

Teknikprogrammet

Gymnasiearbete 100 p

HT 2022 - VT 2023

Schackmotorer

Hugo Lindström

Handledare: Robin Kastberg

Examinator: Marcus Johansson

# Abstract

How does a chess engine work? What different methods are there to calculate the best move in any chess position? The purpose of this text is to get a deeper understanding of chess engines and how different programming techniques and methods affect the outcome of a program. To get a deeper understanding of chess engines I decided on implementing one myself. It turns out every line of code matters in the world of chess engines. Chess engines have to be implemented in a way that allows the program to see as far into the future as possible. This requires an efficient implementation of everything. This optimization is what makes the best chess engines so much better than the rest.

Different methods of implementing a chess engine does not impact performance as much as one might think. A lot of matches between two of the best chess engines in the world, Stockfish and AlphaZero, shows similar levels of performance. The two has a history of being named the best chess engine in the world even though Stockfishes and AlphaZeros method of calculating the best move is completely different.

Innehåll

[Abstract 2](#_Toc126317285)

[Syfte och frågeställning 4](#_Toc126317286)

[Teori 4](#_Toc126317287)

[Schack 4](#_Toc126317288)

[Bitmanipulation 5](#_Toc126317289)

[AND 5](#_Toc126317290)

[OR 5](#_Toc126317291)

[SHIFT 6](#_Toc126317292)

[NOT 6](#_Toc126317293)

[Bitmaps 6](#_Toc126317294)

[Minimax algoritmen Alpha-beta pruning 7](#_Toc126317295)

[Maskininlärning 7](#_Toc126317296)

[Metod och material 7](#_Toc126317297)

[Metod 7](#_Toc126317298)

[Material 7](#_Toc126317299)

[Resultat 7](#_Toc126317300)

[Diskussion 7](#_Toc126317301)

[Slutsats 7](#_Toc126317302)

[Källförteckning 7](#_Toc126317303)

# Syfte och frågeställning

På vilket sätt gör man schackmotorer? Kan man göra det på flera sätt? Vad har olika metoder och tekniker för effekt på resultatet? Det vill jag få reda på. Syftet är att få en djupare förståelse inom programmering och olika teknikers påverkan på resultatet.

# Teori

## Schack

Schack är ett brädspel där två spelare spelar. Spelet går ut på att man tar sin motspelares kung. Den som först tar kungen vinner. Schack spelas på ett 8x8 rutor stort bräde och har 6 olika typer av pjäser. De olika pjäserna kan gå åt olika håll och olika mycket för att ta sig fram genom brädet.[[1]](#footnote-1)

Den enklaste pjäsen är drottningen. Drottningen kan gå rakt eller diagonalt, så långt hon vill, fram till att hon kommer fram till kanten eller en pjäs. Om pjäsen är motståndaren kan hon ställa sig på rutan och därmed ta motståndarens pjäs ur spelet, och om det är hennes eget lag kommer hon behöva stanna på rutan innan. En annan pjäs är tornet. Tornet är en pjäs som bara kan röra sig rakt. Tornet kan röra sig hur långt som det vill, tills den når kanten eller en annan pjäs. Om den kommer till en annan pjäs fungerar den likadant som drottningen. Löparen fungerar likadant som tornet fast den kan bara gå diagonalt. Hästen eller springaren som den också kallas kan bara hoppa fram i en form utav ett L, två steg åt en rak riktning och ett steg åt en annan rak riktning. Det måste dessutom bildas en 90 graders vinkel mellan riktningarna, därmed formen av ett L. Hästen är också den enda pjäsen som kan hoppa över andra pjäser. Hästen kan inte gå till en ruta där en egen pjäs står men kan gå till rutor med motståndarpjäser och då tar de. [[2]](#footnote-2)

Bonden är den pjäs med flest specialregler. Pjäsen kan gå ett steg rakt framåt, men bara om ingen annan pjäs står på rutan framför, inte ens motståndarens. För att en bonde ska kunna ta en annan pjäs behöver en motståndares pjäs stå en ruta diagonalt framför bonden. Då kan bonden ta pjäsen genom att gå dit. Första gången en bonde rör sig framåt kan den ta två steg rakt fram, men det finns en nackdel att gå två steg på en gång. Om man hamnar sida mot sida mot motspelarens bonde kan motståndaren ta bonden genom att gå ett steg diagonalt framåt framför bonden. Detta drag kallas för en passant och kan bara genomföras draget efter en bonde flyttat två steg framåt. När en bonde kommer fram till slutet av brädet kan bonden inte röra sig längre. I stället blir bonden en valfri annan pjäs som inte är en bonde eller en kung. Att uppgradera en bonde till en dam är det enda sättet att få mer än en dam samtidigt på brädet i schack.[[3]](#footnote-3)

Kungen är den viktigaste pjäsen i schack. När en kung dör är spelet slut. När en pjäs hotar att ta kungen, kan ta kungen nästa drag om inte motspelaren spelar någonting, kallas det schack. Om en kung är i schack måste spelaren med en hotad kung flytta på en pjäs så att kungen inte är hotad. Om det inte finns något drag som leder till att kungen blir tagen nästa drag kallas det schack matt. När schackmatt är uppnått är spelet slut. Detta betyder att kungen är den enda pjäsen som inte kan bli tagen. Om en spelares alla drag skulle leda till att kungen blir tagen nästa drag men kungen inte är i schack uppnås patt. Patt innebär att spelet blir oavgjort. Precis som bonden har kungen ett specialdrag. Om kungen inte har gått än, något eller båda av tornen inte har gått än samt att det inte finns några pjäser mellan ett torn som inte gått än och kungen kan kungen flytta två steg mot tornet och tornet flyttar till rutan mellan där kungen stod från början och där slutar efter att den flyttats.[[4]](#footnote-4)

När människor pratar om brädet och hur pjäser flyttas brukar man använda det som kallas för algebraisk notation. Den funkar genom att dela upp brädet i ett koordinatsystem. Varje vertikal rad, fil, får en bokstav och varje horisontell rad, rad, får ett nummer. Sedan refererar man till varje ruta som vilken fil den ligger på och vilken rad den är på. Till exempel är f3 rutan på den sjätte filen från vänster och den tredje raden. För att representera pjäser använder man ”B” för löpare, ”N” för häst, ”R” för torn, ”Q” för drottning och ”k” för kung. Bönder får ingen bokstav utan ”e4” betyder helt enkelt att en bonde går till ruta e4. För att visa att en pjäs tar en annan används ”x”. För att visa rockad används ”0-0” för kort (på kungens sida), eller ”0-0-0” för långt (drottningens sida). Schack visas med ”+” och schackmatt visas med ”#”.[[5]](#footnote-5)

## Bitmanipulation

Bitmanipulation är en process där man ändrar binära tal genom att använda bitvisa operationer. Bitvisa operationer är en typ av räknesätt som inte liknar någon av våra vanliga räknesätt såsom plus eller minus. Bitvisa operationer bygger på att man går igenom ett till två tal bit för bit (siffra för siffra i binär form) och beroende på operation får du olika resultat. De vanligaste operationerna är AND, NAND, OR, NOR, XOR, NOT, LEFTSHIFT, RIGHTSHIFT, LEFTROTATE och RIGHTROTATE.[[6]](#footnote-6)

AND  
AND (&) jämför två lika stora tal (lika många bitar, även nollor framför). Om båda talen vid position x har en etta kommer det bli en etta vid position x i resultatet. NAND (NOT AND) gör motsatsen, om båda talen vid position x har en etta blir det en nolla vid position x i resultatet, annars en etta.[[7]](#footnote-7)

|  |  |  |
| --- | --- | --- |
| Operation | AND | NAND |
| Tal 1 | 1011101101111111 | 1011101101111111 |
| Tal 2 | 0101100010111010 | 0101100010111010 |
| Resultat | 0001100010011010 | 1110011111000101 |

OR  
OR (|) jämför två lika stora tal och kollar om någon utav talen har en etta vid position x. Om ett eller båda av talen har en etta vid position x får resultatet vid position x en etta. NOR (NOT OR) gör motsatsen. För att skriva NOR använder man !(tal 1 |tal2). Om något eller båda talen vid position x har en etta blir det en nolla i resultatet vid position x. XOR (EXCLUSIVE OR, ^) kollar om ett av talen har en etta och det andra har en nolla. Om ett av talen vid position x har en etta blir resultatet en etta men om båda har en etta eller en nolla blir resultatet en nolla.[[8]](#footnote-8)

|  |  |  |  |
| --- | --- | --- | --- |
| Operation | OR | NOR | XOR |
| Tal 1 | 1011101101111111 | 1011101101111111 | 1011101101111111 |
| Tal 2 | 0101100010111010 | 0101100010111010 | 0101100010111010 |
| Resultat | 1111101111111111 | 0000010000000000 | 1110001111000101 |

### SHIFT

Det finns två olika typer av skiftningar, skiftningar och rotationer. Skiftningar och rotationer tar in två tal och ”skiftar” det första talet så mycket som det andra talet säger. Både rotationer och skiftningar har två varianter, LEFT och RIGHT. I programmering används << för LEFTSHIFT och >> för RIGHTSHIFT medan rotationer inte så ofta har ett tecken då det inte är lika vanligt. Om ett tal skiftas med 1 till vänster läggs en nolla till längst bak i talet (position värd 1) och resten av alla bitar flyttas ett steg åt vänster (varje bit blir värd som nästa position) och eventuellt överflöd kastas antingen bort eller kraschar programmet beroende på programmeringsspråk. Att skifta ett tal till vänster är detsamma som att multiplicera talet med 2^n där n är mängden du skiftar. Skiftar till högre fungerar på samma sätt men åt andra hållet. En nolla läggs till längs fram i talet och resten flyttas en position åt bakåt (blir värda en position mindre).Att skifta till höger betyder Rotationer bygger på samma princip som skifts men istället för att lägga till nollor hamnar överflödet i början eller slutet av talet beroende på riktning. LEFTSHIFT kan också användas för att fylla ett tal med X antal ettor i rad. Detta görs genom (1 LEFTSHIFT X) - 1 då man får X antal ettor. För att fylla ett tal med 64 ettor måste man göra ((1 << 63) – 1 + (1 << 63)). Det går inte att göra ((1 << 64) – 1) då (1 << 64) är större än 64 bitar och skulle därmed leda till ett överflöd.[[9]](#footnote-9)  
 LEFTSHIFT RIGHTSHIFT

En bild som visar text

Automatiskt genererad beskrivning En bild som visar text

Automatiskt genererad beskrivning

LEFTROTATE RIGHTROTATE

En bild som visar text

Automatiskt genererad beskrivning En bild som visar text

Automatiskt genererad beskrivning

Alla bilder visar SHIFT / ROTATE med det andra talet som 1.

### NOT

NOT inverterar alla bits av ett tal det vill säga byter ut alla nollor mot ettor och ettor mot nollor.[[10]](#footnote-10)

|  |  |
| --- | --- |
| Operation | NOT |
| Tal | 0101100010111010 |
| Resultat | 1010011101000101 |

## Bitmaps

Då moderna datorer sparar tal som 64 bitar och att ett schackbräde har 64 rutor gör att det passar väldigt bra att representera brädet i binära tal. För att visa att en pjäs finns på en ruta får rutans position i talet ett värde av 1 och annars 0. Dessa tal kallas för bitmaps. Man använder en bitmap för varje typ av pjäs samt en för varje färg. Om man vill veta om det finns en vit häst på f6 kan man helt enkelt ANDa bitmapen för de vita pjäserna och bitmapen för hästar och kolla om biten på plats 45 (f6) är påslagen. Om man ANDar två bitmaps får man en ny bitmap. I fallet med vita hästar får man en bitmap där en etta betyder att det står en vit häst där. För att kolla om en viss bit är aktiverad kan man ANDa bitmapen med 1 LEFTSHIFT biten och kolla om talet är större än noll. Om man till exempel vill veta om det är en vit häst finns på f6 kan man ANDa bitmapen för vita hästar (vita\_pjäser AND hästar) med 1 LEFTSHIFTAT med 45 (f6). Sedan kollar man bara om det blir större än noll. Det hela blir alltså (((vita\_pjäser & hästar) & (1 << 45)) > 0).[[11]](#footnote-11)

Om man vill flytta en pjäs från en ruta till en annan måste man ta bort pjäsen från den gamla rutan och lägga till den på den nya. Man måste dessutom ta bort en eventuell motståndares pjäs. För att ta bort en pjäs kan man fylla ett tal med ettor ((1 << 63) – 1 + (1 << 63)), ta bort originalrutan (tal\_med\_ettor – (1 << ruta)) och sedan ANDa med pjäsens färg och typ. För att lägga till en pjäs tar man pjäsens färg samt typ och ORar med (1 << rutan). Sedan måste man ta bort en eventuell motståndares pjäs på samma sätt som man tog bort sin egen. Om motståndarens pjäs är av samma typ som din ska man inte ta bort typen utan bara färgen. Om man till exempel tog en bonde med en bonde kan inte programmet ta bort en bonde där motståndaren stod eftersom din bonde nu står där.[[12]](#footnote-12)

Man kan också representera andra saker med hjälp utav bitmaps. Till exempel kan man spara om och var varje person kan göra rockad med ett fyra bitars långt nummer. Om första biten är en etta kan vit göra kort rockad, om andra biten är en etta kan vit göra lång rockad och så vidare. [[13]](#footnote-13)

## Minimax algoritmen

Minimax algoritmen är en algoritm för en mot en spel. Algoritmen går ut på att man väljer det bästa draget i den nuvarande positionen genom att testa alla kombinationer av drag och se vilken som leder till det bästa resultatet. Detta görs genom att simulera alla möjliga drag för en position och för varje ny position simulerar man alla möjliga drag i den. Man fortsätter så långt det går innan det blir för många positioner och det går för långsamt. När man når slutet kollar man hur bra positionen är för dig. Om draget är bättre än tidigare bästa drag, sparar man resultatet. Sedan väljer man den drag som garanterar det bästa resultatet. Det bästa resultatet är alltid det som är minst bra för motståndaren. Algoritmen heter minimax eftersom du letar efter drag som är så bra som möjligt för dig (max), sedan så bra som möjligt för motståndaren (min), sedan så bra som möjligt för dig, och så vidare. [[14]](#footnote-14)

När man kör algoritmen får man ett träd av noder och kanter där noder är punkter av data och kanter är kopplingar mellan noder. I det här fallet representerar noder positioner och dess värdering och kanter drag.[[15]](#footnote-15)

Diagram

Description automatically generated

Exempel 1: Minimax träd. Cirkel är en nod (position) och linjer är kanter (drag).

Alpha-beta pruning  
Alpha-beta pruning är en algoritm som bygger på minimax men är mycket snabbare. Det visar sig att om vi kommer till en nod som är sämre än tidigare behöver vi inte kolla längre ner i den grenen. I exemplet ovan behöver vi inte gå igenom nod 10. Nod 4 letar efter ett så litet värde som möjligt. Eftersom nod 9 har värde fyra kan inte noden vid min bli värd mer än så. Då nod 1 vill bli värd så mycket så möjligt vet vi att nod 4 inte kommer kunna ge oss ett bättre värde och där med slutar vi simulera drag där.[[16]](#footnote-16)Diagram

Description automatically generated  
Exempel 2: Minimax träd med alpha beta pruning. Nod 10 försvinner då den garanterat inte förbättrar resultatet.

I ovanstående exempel försvinner bara en nod. Detta låter inte som en jättestor optimering. Men då trädet blir både djupare (fler lager) och bredare (fler noder per lager) försvinner fler och fler noder. I bästa fall kan ett träd som innehåller mängd noder minskas till ungefär .[[17]](#footnote-17)

För att optimera ännu längre och få så få noder som möjligt kan man sortera dragen. Om man sorterar drag efter sannolikhet över att vara bra. Till exempel är det ofta bättre att ta en pjäs eller göra schack än att bara flytta en bonde ett steg framåt. Genom att sortera drag kan man hitta de bästa dragen tidigare. Om man hittar ett väldigt bra drag tidigare kan man ta bort fler drag.[[18]](#footnote-18)

Diagram, schematic

Description automatically generated

Exempel 3: Alpha beta pruning med sorterade drag.

I exemplet ovan kunde vi ta bort två noder. Detta är optimalt. Med bredare och djupare träd blir andelen noder som man kan ta bort större.[[19]](#footnote-19)

## Maskininlärning

Maskinlärning är en del av AI där neurala nätverk används för att fatta beslut. Neurala nätverk kan användas på många sätt inom programmering, till exempel att göra en schackmotor. Ett neuralt nätverk är en mängd neuroner som är ihopkopplade. Dessa neuroner kan vara input, output, eller en så kallad gömd neuron. För att fatta ett beslut tar input neuroner in data som sedan skickas in till ett antal gömda neuroner som sedan skickar vidare signaler till output neuroner som i sin tur gör beslutet. Vilken signal gömda neuroner skickar beror på inputen. Genom att träna nätverket på testdata lär nätverket sig vad olika inputs betyder och vad som är en lämplig slutsats.[[20]](#footnote-20)

I en schackmotor kan man använda maskinlärning på många sätt. Till exempel att sortera dragen eller värdering av positioner. Att använda sig av maskinlärning kan leda till mer oförutsägbara schackmotorer som ser brädet på ett nytt sätt. [[21]](#footnote-21)

# Metod och material

## Metod

­­­­­­­­­Jag började med att programmera spelet schack. Det första jag gjorde var att representera brädet. Detta gjorde jag genom att förvara många bitmaps i en rust struct som kallades board. Sedan implementerade jag drag. Jag började med att generera var varje pjäs kunde gå. Sedan gjorde jag så att man kunde spela de genererade dragen.

Nästa steg var GUI (Graphical User Interface). För att göra ett GUI behöver man ett GUI library. Jag bestämde mig för att använda GGEZ för Rust. Jag skapade en ruta men några rektanglar och importerade lite gratis bilder in i spelet. Spelet såg ut som ett schackbräde, men man kunde inte klicka för att röra pjäserna. För att räkna ut vilken ruta musen var över använde jag musens position. Jag fixade bönders befordran genom att rita 4 rutor i mitten med de fyra olika alternativen. För att befordra en bonde gick man till rutan längst upp och tryckte på knappen för vilken pjäs.

Efter att jag hade implementerat ett GUI visade det sig att min implementation av drag inte fungerade. Detta berodde på att jag hade blandat ihop ordningen på när vad hände under draget. Detta ledde till att när en pjäs tog en annan pjäs hände det att pjäsen försvann visuellt då den inte fanns enligt färg-bitmapen men var kvar enligt bitmapen för pjäserna. Det gjorde att pjäser magiskt bytte pjästyp när en pjäs gick på en ruta där en tidigare pjäs varit.

Efter det började jag med AI. Jag började med minimax. Minimax var egentligen inte några problem men jag insåg att min implementation för drag inte var optimal. Tyvärr hade inte jag tid att fixa det så detta förblev ett problem, även i slutet. Det fanns även ett till problem. Motståndaren gjorde inga bra drag. Detta berodde på att jag inte hade sagt åt schackmotorn att det var en bra sak att ha pjäser i mitten. Efter en lång tid av finjustering av värden fick jag ett resultat som prioriterade mitten av brädet men ändå hade en balans. Det sista jag gjorde var att implementera alpha-beta pruning.

## Material

Rust, GGEZ, Clion, Vim.

# Resultat

En schackmotor fungerar genom att se in så många drag som möjligt i framtiden och välja det drag i nuvarande position som ger den bästa utfallet. För att göra detta simulerar datorn miljontals positioner och ger varje position ett värde.

När jag skapade en schackmotor lyckades jag se in tre halva drag (vit, svart, vit) i framtiden. Detta innebär att min maskin inte ens är i närheten av en människa, varken spelmässigt eller i antal drag man kan se in i framtiden. För att komma dit använda jag mig utav minimax och alpha-beta pruning. Anledningen till att jag inte lyckades så bra beror antagligen på att jag inte implementerade schack på ett bra sätt. Dessutom är min funktion för att värdera positioner inte så bra. En schackmotor som är bra implementerad är Stockfish. Stockfish använder sig utav alpha-beta pruning och kan nå upp till 20 halvdrag inom 5 sekunder i en browser. Det finns även andra schackmotorer såsom Leela och AlphaZero. Dessa använder sig av deep learning och är inte i närheten lika snabb när det kommer till mängd positioner men när det kommer till spelet är de ungefär lika bra.

# Diskussion

Att göra en schackmotor är inte lätt. Inte nog med att man måste programmera ett fungerande schackprogram med många små regler såsom rockad och patt, utan man måste också göra ett snabbt och effektivt program så ens schackmotor har en användning. Jag lyckades inte göra ett effektivt program. Detta berodde antagligen på många saker. Projektet var det första stora programmet jag skrev i Rust. Att använda ett nytt programmeringsspråk gör alltid allt lite svårare då man inte vet vilka funktionaliteter språket har. Min implementation av drag var heller inte så bra. Detta kan ha berott på att jag inte spenderade tillräckligt med tid på det men sannolikt använde jag en ineffektiv algoritm. Att jag inte kom på en effektivare algoritm beror på att jag inte gjort något liknande förut och dessutom inte kunde språket jag skrev den i. Att göra en bra schackmotor handlar inte bara en effektiv metod att generera drag. Även andra faktorer som den rekursiva minimax algoritmen samt representation av brädet måste vara effektiva. Då jag inte visste om många funktioner i Rust fick jag skriva mina egna. Detta är ofta inte optimalt.

Olika metoder att göra schackmotorer skiljer sig mycket när det kommer till implementation men mindre när det kommer till resultat. Detta visar Stockfish och AlphaZero. Att göra en schackmotor

# Slutsats

Olika metoder och tekniker att göra en schackmotor kan ge programmet olika typer av färdigheter eller samma färdigheter men med mycket högre precision. Detta beror på att olika tekniker är olika effektiva. Till exempel gör alpha-beta pruning att minimax algoritmen kan söka genom många fler positioner. Detta i sin tur kommer leda till att schackmotorn med alpha-beta pruning implementerat alltid har en större potential än den utan. Men det räcker inte med att bara välja den snabbaste metoden. För att göra en så bra schackmotor som möjligt krävs en otroligt effektiv implementation av själva spelet. Det spelar inte någon roll om du kan sortera alla drag optimalt eller har ett bra neuralt nätverk för att värdera positioner bättre än någon annan på planeten. Om inte programmet kan simulera dragen snabbt nog kommer inte programmet kunna nå sin fulla potential.

# Källförteckning

Chess.com. (24 mars 2019). “Chess Notation – The Language of the Game” <https://www.chess.com/article/view/chess-notation>

Wikipedia. (26 januari 2023). ”Bitwise operation” <https://en.wikipedia.org/wiki/Bitwise_operation>

Chess Programming WIKI. (12 mars 2022). “Bitboards” <https://www.chessprogramming.org/Bitboards>

Chess Programming WIKI. (1 december 2021). “Alpha-Beta” <https://www.chessprogramming.org/Alpha-Beta>

Chess Programming WIKI. (12 mars 2022). “Neural Networks” <https://www.chessprogramming.org/Neural_Networks>

1. Chess.com. (24 mars 2019). “Chess Notation – The Language of the Game” <https://www.chess.com/article/view/chess-notation> [↑](#footnote-ref-1)
2. Chess.com. ”Chess Notation – The Language of the Game” [↑](#footnote-ref-2)
3. Chess.com. ”Chess Notation – The Language of the Game” [↑](#footnote-ref-3)
4. Chess.com. ”Chess Notation – The Language of the Game” [↑](#footnote-ref-4)
5. Chess.com. ”Chess Notation – The Language of the Game” [↑](#footnote-ref-5)
6. Wikipedia. (26 januari 2023). ”Bitwise operation”  
    <https://en.wikipedia.org/wiki/Bitwise_operation> [↑](#footnote-ref-6)
7. Wikipedia. ”Bitwise operation” [↑](#footnote-ref-7)
8. Wikipedia. ”Bitwise operation” [↑](#footnote-ref-8)
9. Wikipedia. ”Bitwise operation” [↑](#footnote-ref-9)
10. Wikipedia. ”Bitwise operation” [↑](#footnote-ref-10)
11. Wikipedia. ”Bitwise operation” [↑](#footnote-ref-11)
12. Wikipedia. ”Bitwise operation” [↑](#footnote-ref-12)
13. Wikipedia. ”Bitwise operation” [↑](#footnote-ref-13)
14. Chess Programming WIKI. (1 december 2021). “Alpha-Beta” <https://www.chessprogramming.org/Alpha-Beta> [↑](#footnote-ref-14)
15. Chess Programming WIKI . ”Alpha-Beta” [↑](#footnote-ref-15)
16. Chess Programming WIKI . ”Alpha-Beta” [↑](#footnote-ref-16)
17. Chess Programming WIKI . ”Alpha-Beta” [↑](#footnote-ref-17)
18. Chess Programming WIKI . ”Alpha-Beta” [↑](#footnote-ref-18)
19. Chess Programming WIKI . ”Alpha-Beta” [↑](#footnote-ref-19)
20. Chess Programming WIKI. (12 mars 2022). “Neural Networks” <https://www.chessprogramming.org/Neural_Networks> [↑](#footnote-ref-20)
21. [↑](#footnote-ref-21)