Hur mycket snabbare är två minimax schackdatorer med optimeringarna alpha-beta pruning, undo move och MVV-LVA move order?

November 2023

Arvid Kristoffersson, Elias Lindstenz

arvid.kristoffersson@elevmail.stockholm.se, elias.lindstenz@elevmail.stockholm.se

Abstract

This paper studies how effective undo move, alpha-beta pruning, and move order optimizations are for the execution time of an array represented and a bitboard represented chess engine. Due to the various applications of the minimax algorithm in game theory, artificial intelligence, planning, and economics, the results are relevant and contribute to understanding the effectiveness of some optimization techniques. The method consists of measuring the execution time of several engines for 20 regular positions, 20 tactical positions, and 20 endgame positions. The results show that move order optimization with alpha-beta pruning is the most effective for reducing execution time, followed by alpha-beta pruning and undo move optimization. Almost all results are statistically significant. The standard deviation for the array represented chess computer with alpha-beta pruning optimization is probably large since the array represented chess computer cannot generate moves heuristically beneficially without cost. A small number of possibly biased positions were tested at a limited search depth, which could significantly decrease accuracy. Future studies should use randomly selected positions and use more optimizations to allow for deeper searches when comparing the different optimizations.

Inledning

Det finns ungefär 10^{120} distinkta schackpartier som kan existera, om man inkluderar pseudolegala schackdrag (schackdrag som följer pjäsers ursprunliga sätt att gå, men som inte är tillåtna). En dator som beräknar ett spel varje mikrosekund skulle därmed ta ungefär 10^{90} år för att göra det första draget (Shannon C. E. 1949). Därmed är schack ett heuristiskt problem. Ett problem som inte går att lösa deterministiskt, utan som löses med smarta gissningar som approximerar. Schack är ett brädspel som är över 1000 år gammalt. Idag är schack ett brädspel som flera miljoner individer är fascinerade av. Med den tekniska revolutionen som har skett under de senaste årtionden har många intresserade schackspelare försökt lösa detta heuristiska problem.

Schackdatorn Deep Blue, byggd av IBM, var den första schackdatorn som lyckades vinna över schackvärldsmästaren Garry Kasparov 1997 (IBM u.å.). Sedan dess har schack lösts på nya sätt med nya metoder och optimeringar. Dagens starkaste schackdator är Stockfish 16 (Stockfish u.å.). Algoritmen som ligger som grund till den heter minimax.

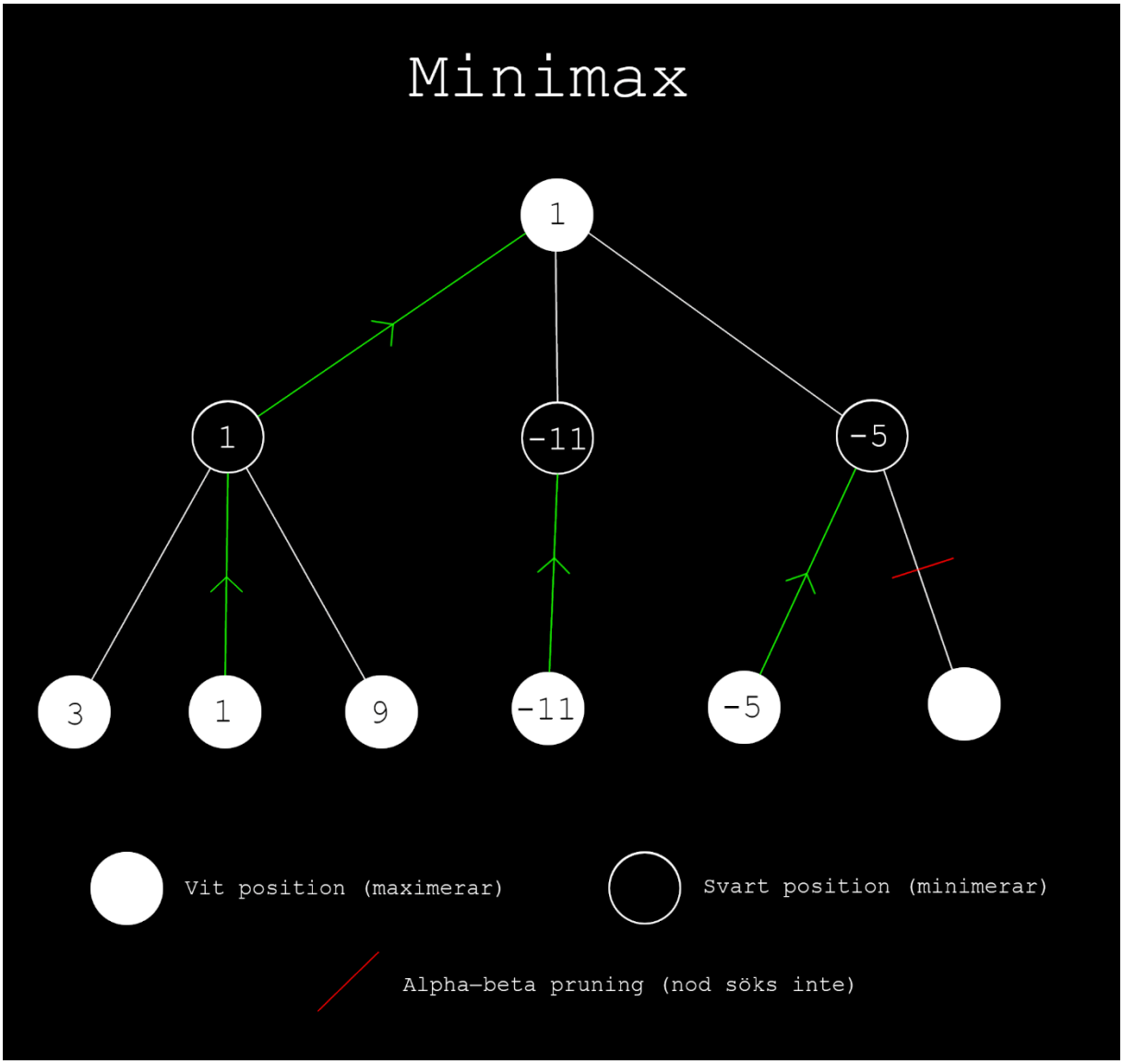

Fig. 1. Minimaxalgoritmen demonstrerad som en graf (sökträd). Sökning sker uppifrån ner och sedan från vänster till höger. Vid den sista svarta noden får vit minst ett och den svarta noden högst minus fem. Därmed behöver inte resterande barnnoder till den sista svarta noden sökas.

Minimax är en heuristisk algoritm som bestämmer det bästa möjliga förutsatt det sämsta möjliga genom att söka genom ett träd. Minimax används i olika spel som Tic-tac-toe och Backgammon, men också inom artificiell intelligens och finans. Exempelvis kan minimax användas för att bestämma optimala bud i double auctions (Shafer R. C. 2020) och planera resursallokering för infrastruktur när det är osäkert vad som kommer hända i framtiden (Anderson E. & Zachary S. 2023). Vi undersökte optimeringar till denna algoritm för att bidra förståelse för effektiviteten av optimeringar. Det är för att effektivisera användningen av minimax inom dessa områden.

Inom grafteori representerar varje schackposition i minimax en nod. Genom att en nod (föräldernod) skapar ett antal barnnoder som är schackpositioner som uppnås av nya drag kan ett schackspel simuleras. Efter ett visst antal drag (ply) in i simuleringen utvärderas positionerna. I minimaxalgoritmen returnerar alla barnnoder ett värde till en föräldernod som väljer den nod som gynnar föräldernoden maximalt. Föräldernoden väljer det maximala eller minimala värdet från de returnerade värdena beroende på om det är vit eller svart som spelar. Detta fortsätter tills den ursprungliga noden (schackpositionen) får ett värde för den bästa serie schackdrag den kan göra för att

vinna förutsatt att motståndaren gör samma sak. Antalet ply schackdatorn simulerar är sökdjupet. Ju större sökdjup desto bättre utvärdering och därmed större möjlighet att vinna får en schackdator. Problemet är att det inte går att söka till schackspelets slut från en viss position eftersom antalet noder ökar exponentiellt för varje extra ply. Från varje schackposition kan en spelare göra ungefär 30 olika drag (Shannon C. E. 1949). För varje djup ökar därmed antalet noder med ungefär 30 gånger. Att söka två drag framåt kräver en genomsökning av ungefär 900 noder medan ett sökdjup på sex ökar antalet noder till ungefär 729 000 000 noder.

Stockfish 16 kan söka till ett djup av mer än 30 med olika optimeringsmetoder. En av de viktigaste av dessa är alpha-beta pruning. Algoritmen sparar värdet på den bästa positionen motståndaren redan kan nå vilket gör att sökningen kan bortse från alla positioner som leder till bättre positioner än det (Knuth D. E. & Moore R. W. 1974). Detta visas förenklat i *fig. 1*. Genom att söka positioner i ordningen som bortser från flest positioner minimeras antalet sökta noder (Bettadapur & Prakash 1986). Detta ger snabbare söktider och därmed möjligheter att söka till större djup (Franklin L. & Malmberg H. 2023). Optimeringen att ordna sökningen benämns move order. En speciell typ av optimeringsmetoden move order heter MVVLVA (Most Valuable Victim – Least Valuable Aggressor). Denna ordning söker pjäser med lägre värde som tar pjäser med högre värde först.

Olika schackdatorer generar drag på olika sätt. Stockfish använder något som heter magic bitboards, som är en mycket snabb metod att generera drag. Andra datorer generar drag traditionellt genom att iterera genom alla pjäser och bestämma vilka drag som är möjliga förutsatt ett schackbräde. Exempelvis kan det verifieras om det finns en pjäs på rutan framför en bonde för att bestämma om bonden får ta ett steg fram. Nackdelen med denna generationstyp är att det kan öka exekveringstiden.

I minimaxalgoritmen modifieras ett schackbräde under sökning. För varje ny nod ändras schackbrädet efter det drag som leder till den nya schackpositionen. Det finns två olika sätt att hantera schackbrädet. Antingen skickar varje nod en modifierad kopia av positionen, eller så skickas samma bräde, som återställs när den returneras. Problemet med att skicka kopior är att datorer tar tid att kopiera schackbräden, speciellt om de tar upp mycket minne i datorn. Ännu en optimeringsmetod, som benämns undo move, är att återställa i stället för att kopiera schackbräden.

Vi undersökte hur snabbt schackpositioner kan sökas igenom med optimeringarna alpha-beta pruning, MVVLVA move order och undo move för att bestämma hur värdefulla de är. Vi undersökte detta för tre olika typer av schackpositioner. Dessa är normala, taktiska och slutspelspositioner. Normala positioner är positioner inom de första 20 ply och där ingen sida tydligt leder eller ingen sida kan vinna material. En taktisk position är en position där en sida kan vinna material genom en serie drag som är mer komplicerad än att endast ta en pjäs som inte skyddas. Exempelvis kan en taktisk position vara en position där en löpare kan röra sig till en ruta som attackerar två pjäser samtidigt och där motståndaren endast kan skydda ena pjäsen. Slutspelspositioner är positioner med 15 eller färre pjäser. En ytterligare nyans vi undersökte var hur dessa skiljer sig för två olika typer av brädesrepresentationer. Ena är uppbyggd av en 8x8 matris med signerade 32-bit heltal (benämns AR). Den andra är uppbyggd av 13 osignerade 64-bit heltal (benämns BB). Signerade betyder att talen kan vara både positiva och negativa, medan osignerade betyder att de endast kan vara positiva. För tolv av dessa representerar heltalet en typ av pjäs och varje bit en rut. Ett av heltalen innehåller istället extra information som exempelvis om rockad är tillåtet och på vilken ruta som nästa spelar får göra en passant. Bitboard schackdatorn använder traditionell draggenerering.

Metod

Med C++ high resolution clock mätte vi tiden det tog för en minimax schackdator med sökdjup fyra att beräkna 20 normala positioner, 20 taktiska positioner och 20 slutspelspositioner (bilaga 1). Detta replikerade vi åtta gånger med olika schackdatorer. Schackdatorerna var av fyra olika typer på array brädesrepresentation respektive bitboard brädesrepresentation. Typerna var en utan optimeringar och resterande med optimeringarna undo move, alpha-beta pruning och alpha-beta pruning med MVVLVA move order. Alla tester genomförde vi på samma maskin (bilaga 2) efter en omstart. Testerna fick använda minst 99.5% av processorn. För varje schackdator och position exekverade vi 100 tester, beräknade en medelexekveringstid och en relativ skillnad ($1 - \frac{T}{B}$, T = schackdatorns medeltid, B = motsvarande icke optimerade schackdators medeltid) till samma position för motsvarande icke optimerade schackdator. För varje typ av schackposition (normal, taktisk, slutspel) och schackdator beräknade vi standardavvikelser ($\sqrt{\frac{\sum(x-\mu)^2}{N}}$, x = tid för ett test, μ = medelvärde för tester, N = antal tester). Vi beräknade sedan medelvärden för relativa skillnader och medelvärden för standardavvikelser ($\frac{\sqrt{\sum x^2}}{N}$, x = standardavvikelse, N = antalet standardavvikelser) för varje schackdator. Dessa standardavvikelser använde vi för att bestämma signifikansen av de genomsnittliga relativa skillnaderna per schackdator. Vi skrev koden för alla tester i C++. Se bilaga 2 för kompileringsspecifikationer, bilaga 3 för kod och bilaga 4 för förenklad pseudokod.

Resultat

De relativa hastighetsändringarna samt deras standardavvikelser redovisas i *fig. 2* för AR schackdatorerna och i *fig. 3* för BB schackdatorerna. Resultaten visar att de relativa hastighetsskillnaderna för AR schackdatorerna var ~0.30, ~0.50 och ~0.74 för optimeringarna undo move, alpha-beta och alpha-beta samt move order. De relativa hastighetskillnaderna för BB schackdatorerna var ~-0.06, ~0.84 och ~0.99 för optimeringarna undo move, alpha-beta och alpha-beta samt move order. Alla skillnader för AR och BB schackdatorerna är signifikanta med varandra förutom AR schackdatorn med optimeringen alpha-beta. Den är inte statistiskt signifikant med AR move order och AR undo move schackdatorerna. Optimeringen undo move gjorde exekveringstiden långsammare för BB schackdatorn och snabbare för AR schackdatorn. Optimeringarna alpha-beta och alpha-beta samt move order hade större effekt för BB schackdatorerna än AR schackdatorerna. Se bilaga 5 för specifik data.

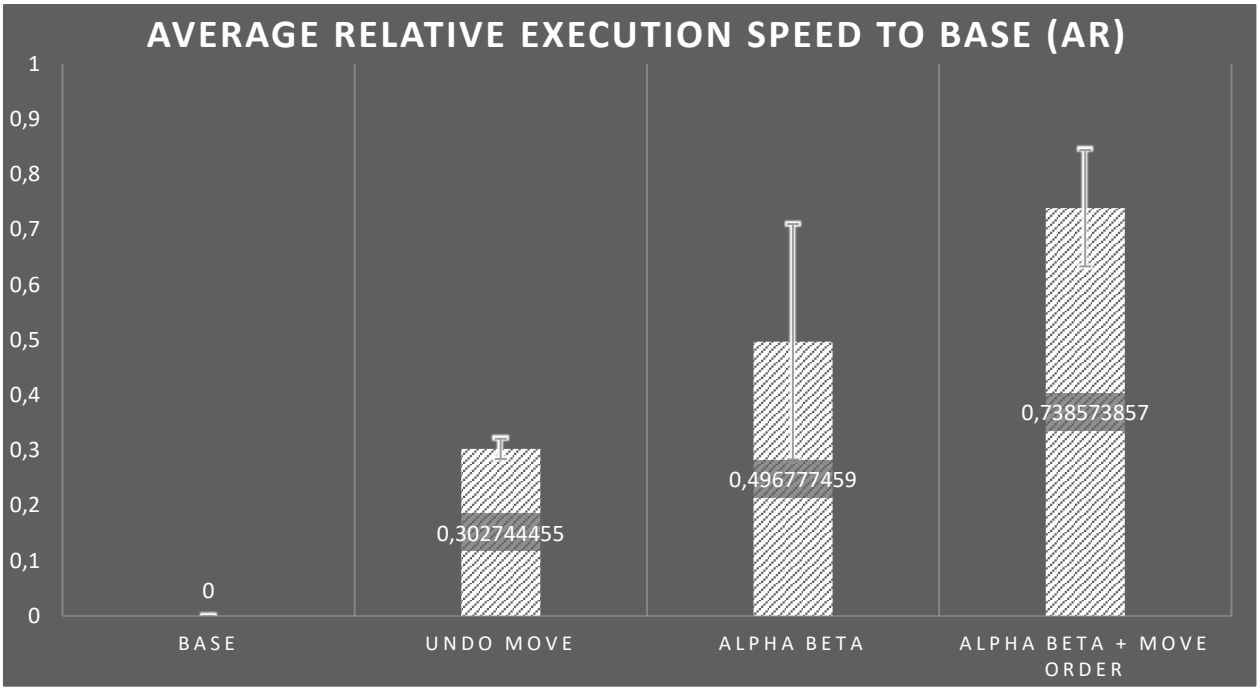

Fig 2. Average Relative Execution Speed to Base (AR). Relativa hastighetsökningar för AR schackdatorer med optimeringarna undo move, alpha-beta och alpha-beta samt move order. Värdet noll indikerar ingen hastighetsskillnad. Värdet ett indikerar en oändlig hastighetsökning.

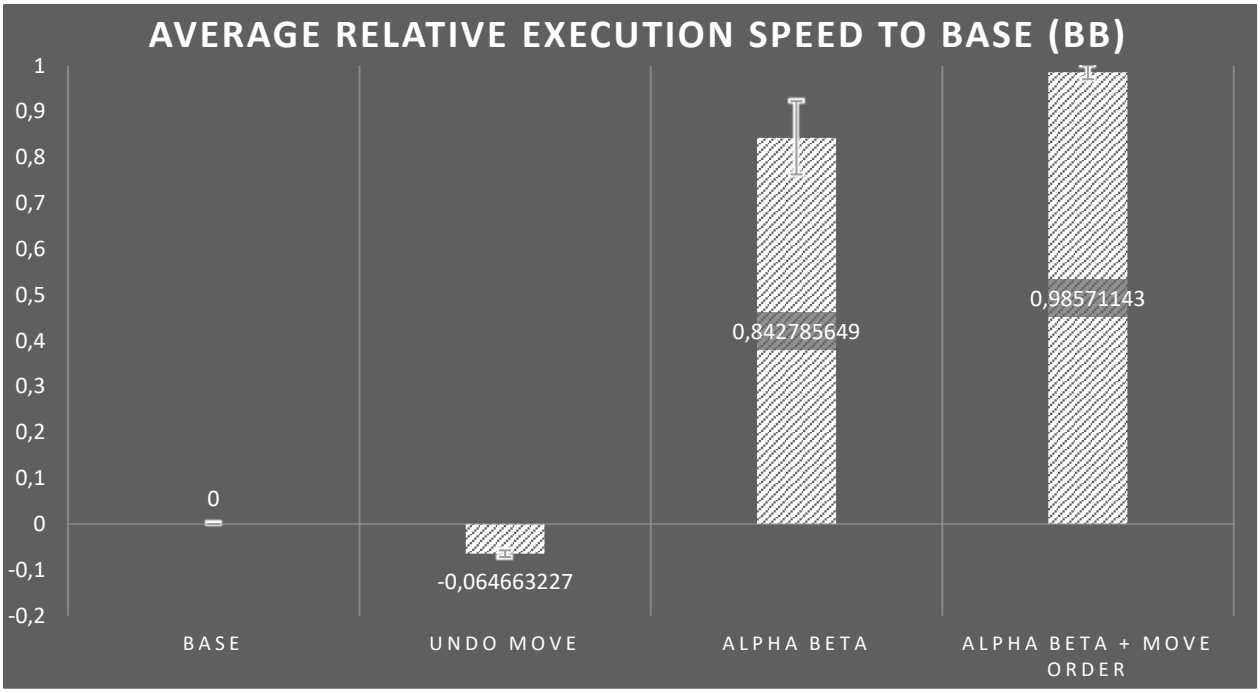

Fig 3. Average Relative Execution Speed to Base (BB). Relativa hastighetsökningar för BB schackdatorer med optimeringarna undo move, alpha-beta och alpha-beta samt move order. Värdet noll indikerar ingen hastighetsskillnad. Värdet ett indikerar en oändlig hastighetsökning.

Diskussion

Resultatet visar att optimeringen alpha-beta tillsammans med move order ger den största hastighetsökningen för både BB och AR schackdatorerna. Efter det kommer optimeringen alpha-beta och sist undo move.

För AR schackdatorn är inte alpha-beta pruning statistisk sämre eller bättre än move order eller undo move. En trolig anledning till den stora standardavvikelsen är att AR schackdatorn är i grunden heuristiskt annorlunda. I en BB schackdator är redan alla pjästyper sorterade i olika schackbräden. Det går därmed att generera drag i en heuristiskt fördelaktig ordning (lägst värda pjäser först) utan kostnad. För AR schackdatorn är alla pjäser i samma schackbräde vilket omöjliggör det att generera i stigande ordning av pjäsvärde utan extra tidskostnad. Detta gör att AR schackdatorn ibland kan ha mindre fördelaktiga sorteringar av drag. Detta gör att färre noder kan tas bort av alpha-beta pruning och att exekveringstiden inte minskar lika mycket i vissa positioner.

Optimeringen undo move för BB schackdatorn ökar exekveringstiden medan den minskar exekveringstiden för AR schackdatorn. Detta beror troligen på att det går snabbare att kopiera ett bitboardbräde bestående av 13 64-bit heltal, än att kopiera ett AR bräde bestående av 64 32-bit heltal. Den ökade komplexiteten för att återställa drag tar alltså längre tid än att kopiera bitboardbräden. Ytterligare en möjlig förklaring till den längre exekveringstiden är att komplexiteten för optimeringen undo move skapar flera möjliga utfall för koden. Detta ökar sannolikheten för branch-missses i processorn vilket kan öka exekveringstid. Kompilatorn är också sannolikt optimerad för att kopiera tal.

Resultatet har hög precision eftersom vi exekverade ett stort antal tester på samma maskin. Mätfel är försumbara eftersom high resolution clock i C++ mäter ticks med nanosekunder som upplösning. Riktigheten av resultatet är svårare att bestämma. Det är inte säkert att schackdatorerna är representativa för andra schackdatorer. Exempelvis skulle en magic bitboard schackdator kunna ha olika relativa skillnader. De exakta relativa skillnaderna är mindre riktiga än ordningen av hur effektiva optimeringarna är. Fortsättningsvis är antalet positioner som mättes förhållandevis lågt. Positionerna handplockades också vilket kan ha introducerat en subjektiv faktor eftersom vi kan ha omedvetet valt positioner baserat på vad vi trodde var ”bra” positioner. Trots att normala, taktiska och slutspelspositioner mättes för att representera riktiga schackspel i större utsträckning är det inte säkert att urvalet av positioner är representativt för vanliga schackpositioner. Ännu en möjlig negativ faktor är det begränsade sökdjupet. Det är möjligt att vit eller svart kunde precis ta en pjäs som var skyddad vid flera sista sökdjup och att datorn inte identifierade att motståndaren kunde ta tillbaka, när moståndaren egentligen kunde det. Detta skulle ändra utvärderingen av positionen och påverkar därmed resultatet.

Ytterligare uppmanas läsaren att vara kritisk mot jämförelser mellan AR och BB schackdatorerna. Implementationen för AR och BB schackdatorerna är i mycket stor utsträckning lika bortsett från att operationerna för att hantera schackspel är annorlunda för bitboard och array representationer. Exempelvis kan en AR schackdator flytta en pjäs med två indexeringar i en two-dimensionell array medan en BB schackdator använder bitoperationer för att göra samma sak. Detta är naturliga skillnader mellan de två olika typerna av schackdatorer. Men, det finns vissa implementationer som särskiljer sig i metod utöver de mer fundamentala skillnaderna. Exempelvis behöver BB schackdatorn kalla en extra funktion (”GetMoveCaptures()”) för att move order ska fungera. AR schackdatorn kräver också extra operationer för att indexera MVVLVA matrisen. Det är svårt att bestämma om detta ska tolkas som naturliga skillnader mellan AR och BB schackdatorer eller om det är ett systematiskt fel. Jämförelser mellan optimeringar för samma typ av schackdator har därmed lite större trovärdighet än jämförelser mellan AR och BB schackdatorer.

Om en liknande undersökning ska genomföras bör flera positionerna slumpmässigt plockas från databaser i stället för att handplockas. Detta minimerar risken för att positioner inte representerar

riktiga schackspel. Ytterligare bör schackdatorer med flera optimeringar användas så att sökdjupet inte är lika begränsat och så att schackdatorerna närmare representerar riktiga schackdatorer. Null-move pruning, transposition table, killer moves och magic bitboard draggenerering är exempel på sådana optimeringar.

Undersökningens resultat kan användas för att förstå vilka optimeringar är effektiva och i viss mån också hur effektiva. Det är exempelvis tydligt att optimeringen move order ger en mycket stor effekt. Resultaten kan också användas för att optimera annat som använder minimax. Många situationer där två sidor försöker maximera sig själva och minimera motståndaren kan använda sig av minimaxalgoritmen och dess optimeringar. Exempelvis är det många spel som exempelvis Tic-tac-toe och Backgammon som kan utnyttja minimax. Det finns ytterligare minimax appliceringar inom artificiell intelligens och finans. Det är exempelvis möjligt att bestämma optimala bud i double auctions och att planera resursallokering för infrastruktur när det är osäkert vad som kommer hända i framtiden.

Referenser

Anderson E., ZacharyS. (2023). Minimax decision rules for planning under uncertainty: Drawbacks and remedies. *European Journal of Operational Research* 311(2): pp. 789-800. https://doi.org/10.1016/j.ejor.2023.05.030 (hämtad 2023).

Bettadapur, Prakash. (1986). Influence of Ordering on Capture Search. *IOS Press* 9(4): pp. 180-188. doi:10.3233/ICG-1986-9403 (hämtad 2023).

Franklin L., Malmberg H. (2023). *Optimization Areas of the Minimax Algorithm*. Degree project, KTH. https://kth.diva-portal.org/smash/get/diva2:1778372/FULLTEXT01.pdf (hämtad 2023).

IBM. (u.å.). *Deep Blue*. https://www.ibm.com/history/deep-blue?mhsrc=ibmsearch_a&mhq=deep%20blue (hämtad 2024).

Knuth D. E., Moore R. W. (1974). An analysis of alpha-beta pruning. *Artificial Intelligence* 6(4): pp. 293-326. https://doi.org/10.1016/0004-3702(75)90019-3 (hämtad 2023).

Shafer R. C. (2020). Minimax regret and failure to converge to efficiency in large markets. *Games and Economic Behavior* 124: pp. 281-287. https://doi.org/10.1016/j.geb.2020.07.010 (hämtad 2023).

Shannon C. E. (1949). Programming a Computer for Playing Chess. *Philosophical Magazine* 41(314): pp. 256-275. https://doi.org/10.1080/14786445008521796 (hämtad 2023).

Stockfish. (u.å.). *Stockfish 16*. https://stockfishchess.org/ (hämtad 2023).

Bilagor

Bilaga 1: https://github.com/novrion/GYAR/blob/main/bot/data/in (schackpositioner (FEN-koder))

Bilaga 2: https://github.com/novrion/GYAR/blob/main/technical-specification (maskin- och kompileringsspecifikationer)

Bilaga 3: https://github.com/novrion/GYAR/tree/main/bot/src (kod)

Bilaga 4: https://github.com/novrion/GYAR/blob/main/pseudocode (pseudokod)

Bilaga 5: https://github.com/novrion/GYAR/tree/main/bot/data (resultat)