Technická univerzita v Košiciach
Fakulta elektrotechniky a informatiky

Šachový simulátor využitím pokročilých metód umelej inteligencie

Diplomová práca

2022 **Bc. Pavol Kacej**

Technická univerzita v Košiciach
Fakulta elektrotechniky a informatiky

Šachový simulátor využitím pokročilých metód umelej inteligencie

Diplomová práca

Študijný program:	Informatika
Študijný odbor:	9.2.1. Informatika
Školiace pracovisko:	Katedra počítačov a informatiky (KPI)
Školiteľ:	doc. Ing. Juraj Gazda, PhD.

Košice 2022 **Bc. Pavol Kacej**

Abstrakt v SJ

Práca je založená na analýze a porovnaní existujúcich šachových simulátorov. Z týchto znalostí je postupne implementovaný vlastný typ šachového simulátora, ktorý je založený na strojovom učení na existujúcich hrách. Za pomoci hlbokej konvolučnej neurónovej siete sa model učí imitovať správanie zo vzorky hier hraných ľudskými a počítačovými hráčmi. Ku modelu je pridaný algoritmus Minimax, a následne prebieha experimentálne overenie modelu simulovaním hier, a jeho ohodnotenie. Súčasťou práce je aj tvorba grafického rozhrania pre hru šach, ktorá umožňuje používateľom súperiť, či už proti spomínanému modelu, alebo v hre pre dvoch hráčov.

Kľúčové slová v SJ

strojové učenie, neurónová sieť, konvolučná neurónová sieť, CNN, šach, python, pygame, programovanie, umelá inteligencia, vyhľadávanie, vyhľadávanie v strome, minimax, imitačné učenie

Abstrakt v AJ

The thesis is based on analysis and comparison of the existing chess engines. Based on the knowledge from the analysis, new type of chess engine is implemented, which is using machine learning process over existing games. With the help of CNN, the model learns to imitate behavior from games played by both, human and AI players. Subsequently, Minimax algorithm is added to the solution and the model is verified by experiment; by simulating the games and evaluation. The additional part of the thesis is the creation of user interface for the game of chess, that will allow users to compete against the mentioned model, or in the two players game.

Kľúčové slová v AJ

machine learning, neural network, convolutional neural network, CNN, chess, python, pygame, programming, artificial intelligence, search, tree search, minimax, imitation learning

Bibliografická citácia

KACEJ, Pavol. *Šachový simulátor využitím pokročilých metód umelej inteligencie*. Košice: Technická univerzita v Košiciach, Fakulta elektrotechniky a informatiky, 2022. 54s. Vedúci práce: doc. Ing. Juraj Gazda, PhD.

64097

TECHNICKÁ UNIVERZITA V KOŠICIACH

FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Katedra počítačov a informatiky

ZADANIE DIPLOMOVEJ PRÁCE

Študijný odbor: **Informatika**
Študijný program: **Informatika**

Názov práce:

Šachový simulátor využitím pokročilých metód umelej inteligencie

The chess simulator based on the application of advanced artificial intelligence algorithms

Študent: **Bc. Pavol Kacej**
Školiteľ: **doc. Ing. Juraj Gazda, PhD.**
Školiace pracovisko: **Katedra počítačov a informatiky**
Konzultant práce:
Pracovisko konzultanta:

Pokyny na vypracovanie diplomovej práce:
1. Analyzovať existujúce metódy na báze strojového učenia pre hru šach.
2. Vytvoriť grafické rozhranie pre hru šach.
2. Porovnať implementované algoritmy strojového učenia využitím vhodne zvolených ukazovateľov v tejto hre.
3. Navrhnúť úpravy vybraných algoritmov za účelom zlepšenia vybraných ukazovateľov v hre.
4. Experimentálne overiť navrhnuté riešenie.

Jazyk, v ktorom sa práca vypracuje: slovenský
Termín pre odovzdanie práce: 22.04.2022
Dátum zadania diplomovej práce: 29.10.2021

prof. Ing. Liberios Vokorokos, PhD.
dekan fakulty

Čestné vyhlásenie

Vyhlasujem, že som záverečnú prácu vypracoval samostatne s použitím uvedenej odbornej literatúry.

Košice, 22.4.2022 .....

Vlastnoručný podpis

Poďakovanie

Na tomto mieste by som rád poďakoval svojmu vedúcemu práce doc. Ing. Jurajovi Gazdovi, PhD. za jeho čas a odborné vedenie počas riešenia mojej záverečnej práce.

Rovnako by som sa rád poďakoval svojim rodičom a priateľom za ich podporu a povzbudzovanie počas celého môjho štúdia.

Obsah

Zoznam obrázkov

Zoznam tabuliek

Zoznam zdrojových kódov

Úvod

Tvorba umelej inteligencie pre strategické doskové hry je stále aktuálny problém, ktorý má svoj pôvod v druhej polovici dvadsiateho storočia. Medzi prvé zmienky o umelej inteligencii v šachu môžeme považovať myšlienky britského matematika Alana Turinga, ktorý je známy vynálezom Turingovho stroja, alebo riešením Enigmy. Už v roku 1947 začal na tvorbe teórií, kedy by mohol počítač hrať proti človeku. O tri roky neskôr vytvoril prvý počítačový algoritmus hrajúci šach. Keďže v tom čase ešte nebol vyvinutý vhodný hardvér, na ktorom by mohol byť algoritmus spustený, Turing sám vykonával úlohu počítača, kedy manuálne vypočítaval ťahy podľa algoritmu v hre proti jeho priateľovi. Napriek tomu, že výpočet jedného ťahu trval desiatky minút, a tento algoritmus zápas prehral, zapísal sa do histórie. Turing zverejnil svoje myšlienky v roku 1953.[1]

Veľkú pozornosť medzi laickou aj odbornou verejnosťou nadobudla oblasť v roku 1996 a 1997, kedy prebiehal prvý a druhý zápas medzi superpočítačom Deep Blue[2], a vtedajším svetovým šampiónom Garrym Kasparovom. Oba zápasy pozostávali zo šiestich hier, a dianie sledovali milióny ľudí z celého sveta. Prvý zápas, ktorý sa konal v Equitable Center v meste New York, vyhral veľmajster Garry Kasparov výsledkom 4 ku 2. V druhom zápase, ktorý sa konal o rok na to, už superpočítač zdolal veľmajstra výsledkom 3½ ku 2½. Išlo o historickú udalosť, často interpretovanú ako boj muža proti stroju. Vytvorili sa hneď dva rekordy - porážka majstra sveta počítačom v hre a v zápase. V pozadí toho všetkého však išlo o dôležitú výpočtovú vedu, ktorá zlepšovala schopnosť počítačov spracovávať komplexné vedecké výpočty. Šlo napríklad o pomoc pri vynájdení nových liekov, tvorbu finančných modelov a analýzu rizík alebo efektívne vyhľadávanie v databáze.

Od historického zápasu, kedy superpočítač vyhral nad človekom prebieha nový druh zápasu - zápas medzi rozličnými umelými inteligenciami, nazývanými aj simulátormi. Existuje niekoľko rebríčkov, kde rôzne firmy porovnávajú svoj šachový softvér s inými takýmito softvérmi. Najnovšie však dominuje nový druh simulátora - založený na strojovom učení. Výskum dokazuje, že výsledky

takéhoto typu algoritmu sa prinajmenšom rovnajú výsledkom klasických simulátorov, ak ich neprevyšujú. Rôzne klasické simulátory, akým je napríklad Stockfish, sa snaží najnovšie implementovať techniky strojového učenia, aby dosiahol tie najlepšie výsledky.

Cieľom práce tak bude analyzovať existujúce šachové simulátory, porovnať ich, a pomocou strojového učenia implementovať vlastný simulátor. Tento simulátor bude založený na neurónovej sieti. Je tak potrebné zvoliť správnu metodológiu, akou bude implementácia prebiehať, určiť typ parametrov pre neurónovú sieť, a vykonať spracovanie vstupných a výstupných dát. Výsledok z neurónovej siete bude možné vizualizovať pomocou vlastného používateľského prostredia, ktoré umožní používateľovi hrať, či už proti samotnému algoritmu, alebo inému používateľovi.

Formulácia úlohy

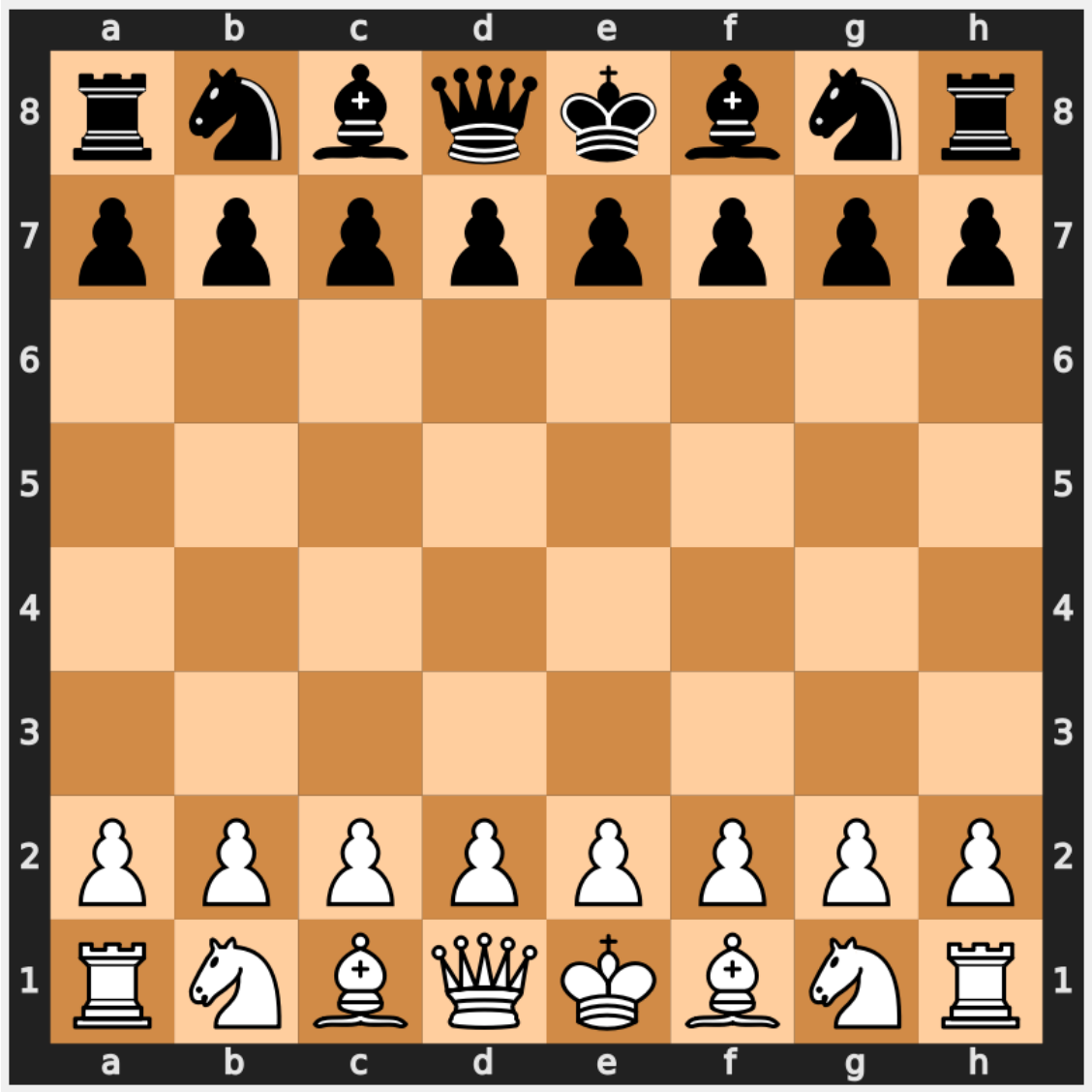
1. Analyzovať existujúce metódy na báze strojového učenia pre hru šach,
2. Vytvoriť grafické rozhranie pre hru šach,
3. Porovnať implementované algoritmy strojového učenia využitím vhodne zvolených ukazovateľov v tejto hre,
4. Navrhnúť úpravy vybraných algoritmov za účelom zlepšenia vybraných ukazovateľov v hre,
5. Experimentálne overiť navrhnuté riešenie.

1 Úvod do šachu

Šach je strategická dosková hra pre dvoch hráčov. Súčasná forma šachu sa sformovala v Južnej Európe počas druhej polovice 15. storočia. Vyvinul sa z pôvodnej Čaturangy, čo je principiálne veľmi podobná hra staršieho pôvodu z Indie. Dnes je šach jednou z najpopulárnejších hier, hranou miliónmi ľudí po celom svete. Patrí medzi abstraktné hry, a nemá žiadnu skrytú informáciu alebo náhodnú informáciu, ako je to napríklad pri kartových hrách.[3] Nachádzajú sa v nej stredoveké prvky, ako napríklad boj medzi armádami, jazdci, alebo ochrana kráľa. Vyrezávanie šachových figúr má svoju dlhoročnú tradíciu, a existuje viacero štýlov ich zhotovenia. Existujú taktiež nálezy historických figúr z 10. storočia, ktoré boli predchodcami tých dnešných. Na druhú stranu, šach má dnes veľkú popularitu v online priestore. V posledných rokoch sa z dôvodu pandémie[4] výrazne zvýšil záujem o online šach.

1.1 Šachovnica a figúry

Hra prebieha na šachovnici, ktorá pozostáva zo 64 štvorcov. Štvorce sú zoradené do mriežky 8x8. Základné postavenie šachovnice je zobrazené na obrázku 1.1. Šach rozlišuje šesť druhov figúr; veža (pozícia `a1`), jazdec (`b1`), strelec (`c1`), dáma (`d1`), kráľ (`e1`) a pešiak (`a2`). Figúry sú na začiatku hry postavené podľa farby, pre každého oponenta symetricky. Pre jednoznačnosť platí, že biela dáma začína na bielom poli, a čierna dáma na čiernom poli oproti. Figúry sa líšia spôsobom pohybu. Vo všeobecnosti platí, že čím je figúra aktívnejšia (má viac možností na pohyb), tým je silnejšia (cennejšia). Dáma a veža sú ťažké figúry, jazdec a strelec sú ľahké figúry. Sila figúr sa môže líšiť podľa pozície. Môže sa tak stať, že tabuľkovo slabšia figúra v aktívnej pozícii bude užitočnejšia ako tabuľkovo silnejšia figúra v pasívnejšej pozícii. Formálne sa pracuje s hodnotením zobrazeným v tabuľke 1.1. Kráľ sa považuje za najcennejšiu figúru v šachu. Figúry konkrétneho hráča sa zvyknú nazývať aj pod súhrnným pojmom materiál.



Obr. 1.1: Základné postavenie figúr

1.2 Podstata a pravidlá

Cieľom hry je dať súperovi mat. "*Šachmat alebo šach-mat alebo mat je situácia v šachu, keď je kráľ ohrozený a po ľubovoľnom ťahu by bol ohrozený tiež. Hráč, ktorého kráľ dostal mat, partiu prehral.*"[5] Ku výhre hráča nad súperom dochádza aj vtedy, keď sa súper vzdá, alebo mu vyprší čas. V prípade, že súperovi vyprší čas, ale nemá dostatok materiálu na to, aby uskutočnil mat, je tento zápas považovaný za remízu.

Remíza môže nastať v niekoľkých hlavných prípadoch. Prvým prípadom je stav, kedy dochádza ku trojnásobnému dosiahnutiu tej istej pozície. Dôvodom môže byť, že hráči opakujú ťahy, pretože tvorba iných ťahov by dala druhému hráčovi výhodu. Druhým prípadom je pozícia, kedy ani jeden z hráčov nemá dostatočný materiál na to, aby dal mat súperovi. Tretím prípadom je pat - stav,

kedy hráč, ktorý je na ťahu nemá žiaden možný ťah, a nie je v šachu (ak by bol v takom prípade v šachu, ide o mat). Štvrtý stav je taký, kedy už ubehlo 50 ťahov bez toho, aby došlo k pohybu pešiaka, alebo odobrania materiálu. Dôvodom je, aby hry neboli príliš zdĺhavé, avšak tento prípad je pomerne vzácny.

V partii začína stále biely hráč ako prvý, a partia prebieha stále po jednom ťahu striedaním bieleho a čierneho. Šachová partia sa delí na otvorenie, ktoré je na začiatku hry a počas neho sa rozvíja pozícia figúr a pešiakov. Nasleduje stredná hra, ktorá je zväčša pozičná a taktická. Poslednou fázou je koncovka, ktorá zvykne obsahovať iba niekoľko figúr.

1.3 Šachová notácia

Šachová notácia slúži na zapisovanie ťahu alebo pozície figúry. Každý štvorec na šachovnici má svoje označenie v algebraickej notácii, ktoré sa skladá z písmena (označujúc stĺpec), a čísla (označujúc riadok). Označenie figúry môže byť pomocou ikony figúry, alebo písmenom podľa konkrétneho jazyka. Slovenská a Anglická notácia figúr je zobrazená v tabuľke 1.1. Notácia pešiaka sa v praxi nepoužíva, keďže je postačujúca jeho koncová pozícia.

Typ:	Pešiak	Jazdec	Strelec	Veža	Dáma	Kráľ
Hodnota:	1	3	3	5	9	∞
Slovenská notácia:	P	J	S	V	D	K
Anglická notácia:	P	N	B	R	Q	K

Tabuľka 1.1: Tabuľka figúr

Notácia ťahu principiálne pozostáva z popisu štartovacej pozície alebo figúry, ktorá sa bude pohybovať, a cieľovej pozície. Príkladom je zápis v Anglickej notácii `Nf3`, ktorý zapisuje ťah jazdca na pozíciu `f3`. Pokiaľ figúra vykonáva branie, môže sa znak × vložiť medzi skratku názvu príslušnej figúry a cieľové pole. Malá rošáda je označovaná ako `0-0`, veľká ako `0-0-0`. Konkrétne pravidlá, akými dochádza k zápisu sú spísané v oficiálnych pravidlách organizácie FIDE[6]. Existuje viacero typov notácií. Medzi najčastejšie používané patria:

- **SAN** (**Standard algebraic notation**) - zápis ťahov štandardizovaný organizáciou FIDE (napr. `Nf3`),
- **LAN** (**Long algebraic notation**) alebo **UCI** - dlhý zápis ťahov zahrňujúc aj štartovaciu pozíciu (napr. `Ng1f3`),
- **PGN** (**Portable Game Notation**) - počítačový zápis šachovej partie s ťahmi v textovom súbore, príklad bude zverejnený v sekcii 6.3,
- **FEN** (**Forsyth–Edwards Notation**) - jednoriadkový počítačový zápis šachovej pozície bez ťahov, napríklad:
  `rn1qkbnr/pbpppppp/1p6/8/2PP4/8/PP2PPPP/RNBQKBNR w KQkq - 1 3`

2 Umelá inteligencia

Pre úvod do problematiky je potrebné pochopiť pojmu šachový simulátor a umelá inteligencia.

Pod pojmom šachový simulátor sa v rámci tejto práce bude nazývať softvér, ktorý je tvorený šachovým hráčom založeným na umelej inteligencii. Jeho snahou je byť v tvorbe ťahov čo najpresnejší, a predčiť tak súpera.

Druhým termínom je spomínaná umelá inteligencia, respektíve umelá inteligencia v doskových hrách. Umelá inteligencia (artificial intelligence) je simulácia ľudskej inteligencie strojom. Stroj je naprogramovaný tak, aby "rozmýšľal", a napodobňoval akcie spôsobom podobným človeku. Termínom sa tiež môže označovať taký stroj, ktorý javí známky spojené s ľudským vedomím, ako učenie alebo riešenie problémov.[7]

2.1 Umelá inteligencia v hrách

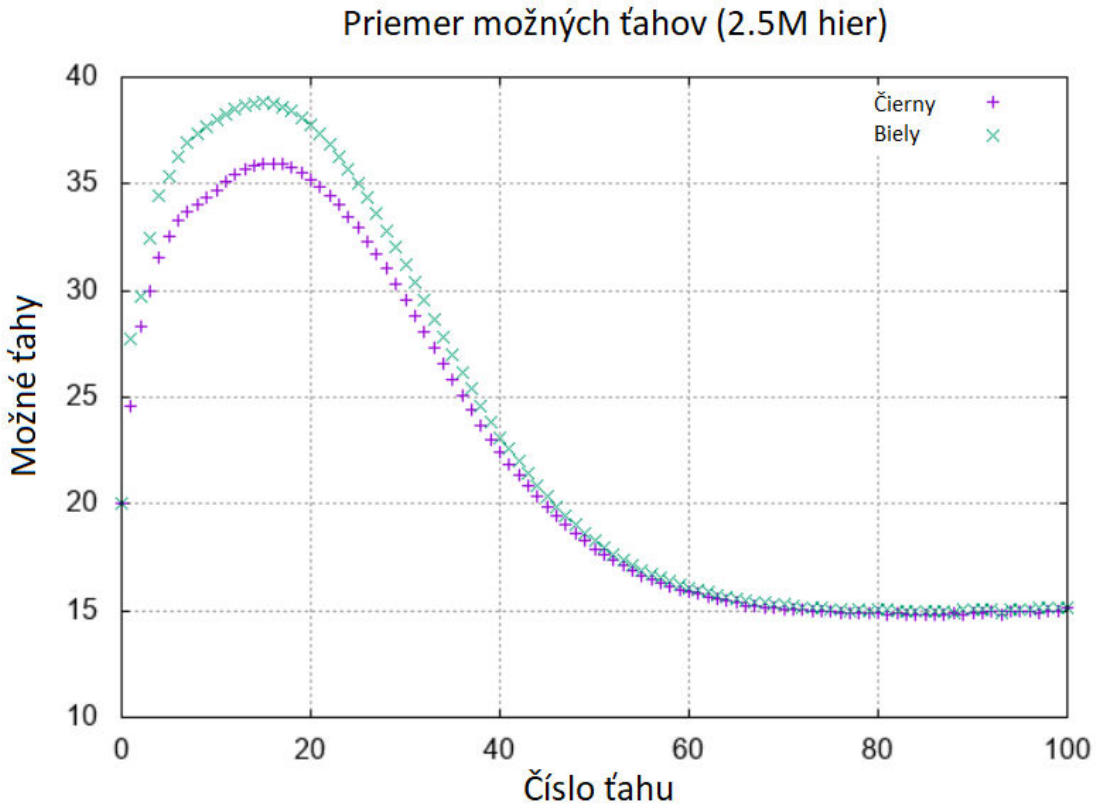
Hry sú dlhodobo považované za ideálny priestor na študovanie umelej inteligencie. Dodnes je veľká časť akademickej práce v oblasti zameraná na tradičné doskové alebo kartové hry, kedy je výzvou výhra nad profesionálnym ľudským hráčom. Tradičné hry sú späté so schopnosťou hráčov manipulovať s hrou pomocou objektov, akými sú figúry na šachovnici, alebo karty v balíku. Pravidlá hry špecifikujú, ako herné objekty interagujú, a fundamentálna kombinatorika rýchlo produkuje masívne herné stromy.[8] Kvôli veľkému nárastu možných akcií je potrebné, aby umelá inteligencia bola schopná robiť správne rozhodnutia. Ich presnosť by mala byť na takej úrovni, aby dokázala konkurovať expertom v konkrétnej disciplíne. Medzi najúspešnejšie počítačové programy minulosti, ktoré pracovali na umelej inteligencii, patrí napríklad Deep Blue. Takéto programy sa zväčša špecifikujú na skúmanie herných stromov technikou hrubá sila, kedy manuálne skúmajú každý herný stav podľa typu vyhľadávania.

2.2 Faktor vetvenia

Množstvo, akým sa dosiahnuteľné pozície zvyšujú, je priamo spojené s faktorom vetvenia danej hry, respektíve priemerným množstvom ťahov v daných stavoch. Faktor vetvenia je možné definovať aj ako počet dcérskych uzlov daného uzla. Na začiatku šachovej partie má biely hráč k dispozícii dvadsať ťahov. Následne môže čierny hráč odpovedať taktiež dvadsiatimi ťahmi. Po ťahu oboch hráčov tak môže vzniknúť 400 možných stavov na šachovnici. Faktor vetvenia pri hre šach sa udáva ako 35. Pre porovnanie, hra Go začína s prázdnou hracou plochou, a čierny hráč môže umiestniť kameň na na jednu z 361 pozícií. Biely hráč môže odpovedať umiestnením svojho kameňa na jednu z 360 pozícií, a tak ďalej. Faktor vetvenia pri hre Go je 250.[9]

Novšie údaje pozostávajúce z analýzy 2,5 milióna šachových partií[10] hovoria o priemernom počte približne 31,1 ťahov na pozíciu. Celkový počet vykonaných ťahov v analýze bol 194 389 820 (76,5 za hru), a celkový počet potenciálnych ťahov bol 6 039 013 721. Tieto zistenia sú zobrazené na obrázku 2.1. Zaujímavým faktom, ktorý je možné vyčítať z grafu, je tiež to, že biely hráč mal stále štatisticky viac možných ťahov pre danú úroveň, ako čierny hráč. Tento rozdiel síce nie je výrazný, avšak je merateľný v jednotkách percent. Vysvetlením by mohlo byť konštatovanie, že biely, výhodou prvého ťahu, má schopnosť tvoriť hru a vytvoriť si tak aktívnejšiu pozíciu. Naopak, čierny hráč je zväčša pasívnejší, a musí reagovať na aktivitu bieleho, čo mierne znižuje počet jeho možností. Ide napríklad o prípad, kedy biely potiahne dopredu s pešiakom, a zablokuje čierneho pešiaka, berúc mu takúto možnosť v nasledujúcom ťahu.

Vyššie faktory vetvenia činia algoritmy, ktoré sledujú každú vetvu v každom uzle, napríklad vyčerpávajúcim vyhľadávaním hrubou silou. Sú výpočtovo drahšie kvôli exponenciálnemu rastu počtu uzlov, vedúcemu ku kombinatorickej explózii. Napriek tomu, že hra Go má väčší faktor vetvenia, pri oboch hrách sa dá sledovať takýto druh explózie. Napríklad ak je faktor vetvenia 10, potom bude 10 uzlov o jednu úroveň nižšie od aktuálnej pozície, 10^2 uzlov o dve úrovne nižšie, 10^3 uzlov o tri úrovne nižšie atď. Čím vyšší je vetviaci faktor, tým rýchlejšie k tejto „explózii“ dochádza. Faktor vetvenia je možné znížiť algoritmom prerezávania.[11]



Obr. 2.1: Faktor vetvenia pre hru šach[10]

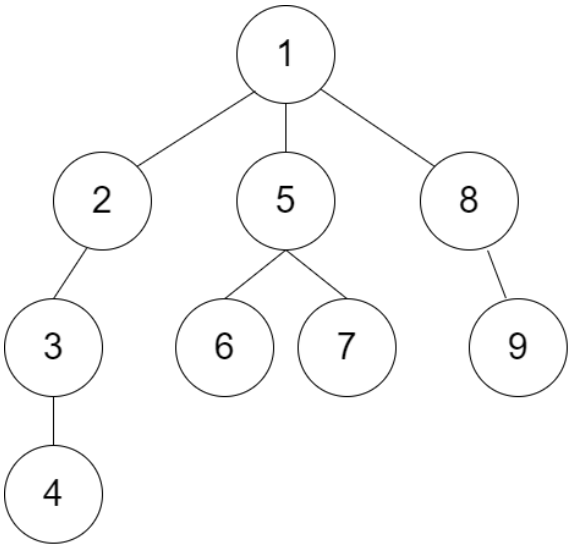
2.3 Vyhľadávacie algoritmy a optimalizácie

Implementácie umelej inteligencie v hrách často vylučujú výpočty, ktoré sú vopred pripravené, a nie je nutné ich znova vypočítavať. V šachu je často využívaná kniha otvorení na začiatku hry, aby sa znížila časovú náročnosť výpočtu. Sú to poväčšine ťahy do ukončenia 2. ťahu (4 polťahy), ale pri komplexnejších otvoreniach ide až o ťahy do ukončenia 7. ťahu (14 polťahov). Ide o tabuľkové otvorenia, ktoré boli desaťročia teoreticky skúmané, a považujú sa za správne (precízne). Ich používanie hráčom často vedie ku výhodnej pozícii konkrétneho hráča.

Existuje aj optimalizácia v koncovej hre. Jednou z nich sú tabuľkové koncovky Syzygy[12], ktoré obsahujú predom vypočítaných viac ako 400 biliárd legálnych pozícií. Všetky tieto pozície boli vypočítané hrubou silou. V prípade, že je na šachovnici 7 alebo menej figúr, je možné využiť práve tieto koncovky. Keďže ich obsah pozostáva z viac ako 18 TB pamäte, prístup k ním je často cez externé alebo cloudové knižnice. Cez tieto knižnice je tiež možné v koncovke zistiť, či je možné vykonať mat v dostatočnom počte ťahov (menej ako 50), alebo či ide o remízovú pozíciu.

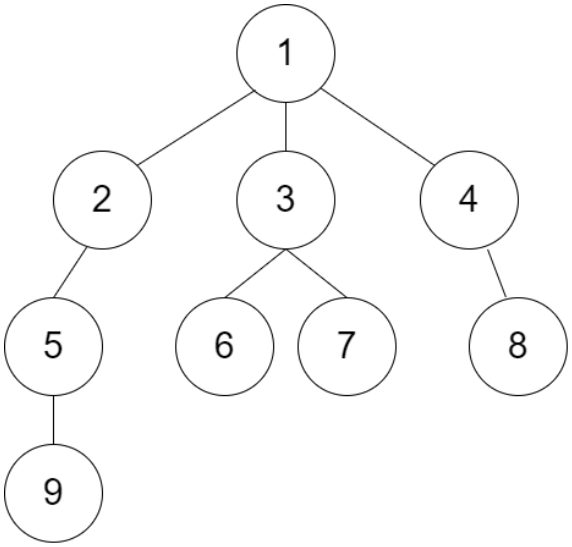
V ostatných prípadoch je používaný jeden z vhodných vyhľadávacích algoritmov. Medzi najčastejšie používané patrí:

- **DFS (Depth-first search)** je vyhľadávanie do hĺbky. Algoritmus začína z koreňového uzla, a postupným vnáraním navštevuje uzly v ďalších hĺbkových úrovniach. Po dosiahnutí danej hĺbky sa pomocou metódy backtracking[13] stále vráti do predchádzajúceho uzla, a opakuje vnorenie pre ďalšie uzly, ktoré ešte neboli navštívené. Poradie, v akom algoritmus navštívi uzly je zobrazené na obrázku 2.2.



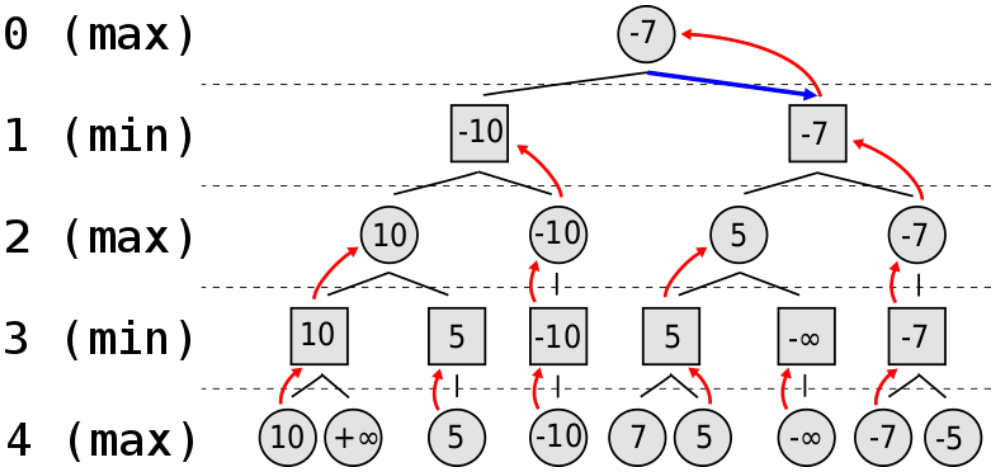
Obr. 2.2: Poradie navštívenia uzlov algoritmom DFS

- **BFS (Breadth-first search)** je vyhľadávanie do šírky. Algoritmus podobne začína z koreňového uzla, avšak po vnorení navštívi všetky uzly v danej (nasledujúcej) hĺbkovej úrovni. V ďalšom kroku sa postupne z každého uzla danej hĺbky vnorí o jednu úroveň nižšie a navštevuje všetky uzly v danej úrovni. V porovnaní s algoritmom BFS je pamäťovo náročnejší, keďže pri jeho implementácii je potrebný zásobník, ktorý uchováva informáciu o tom, ktoré dcérske uzly už boli navštívené. Poradie, v akom algoritmus navštívi uzly je zobrazené na obrázku 2.3.



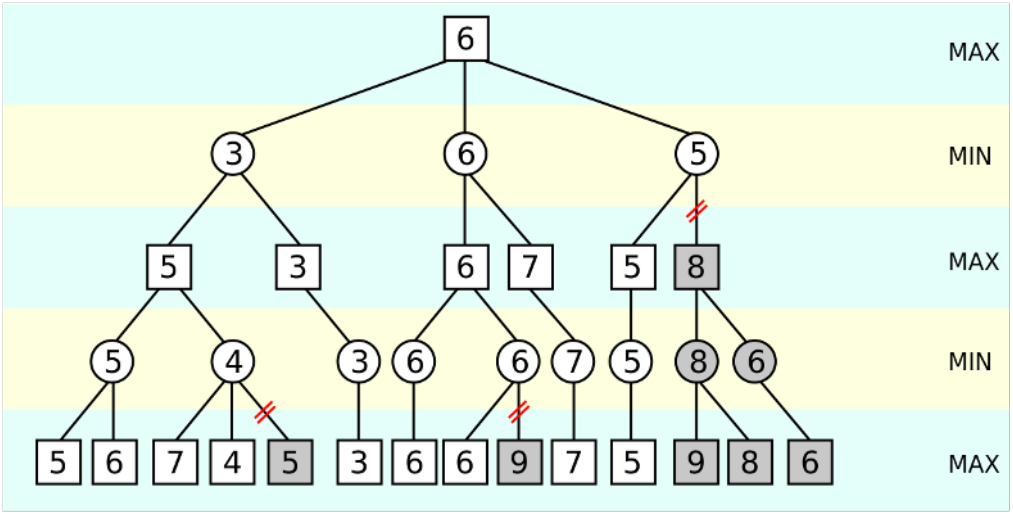
Obr. 2.3: Poradie navštívenia uzlov algoritmom BFS

- **Minimax** je algoritmus, ktorý má určiť najlepšiu stratégiu hry tým, že prezerá dopredu rôzne postupnosti ťahov a snaží sa nájsť takú postupnosť, ktorá vedie k výhre hráča. Minimax algoritmus je považovaný za jeden zo základných algoritmov umelej inteligencie v problematike riešenia problémov opísaných ako hra.[14] Je založený na vyhľadávaní do hĺbky. Pri koncovom uzle je spravidla stav ohodnotený heuristickou funkciou. Následne dochádza k backtracking-u, a po navštívení všetkých dcérskych uzlov daného uzla sa jeho hodnota určí podľa hodnôt v dcérskych uzloch. Hodnota sa určí tak, že sa vyberie najväčšia hodnota dcérskych uzlov (ak je na ťahu biely hráč), alebo najmenšia hodnota dcérskych uzlov (ak je na ťahu čierny hráč). Spôsob, akým nastáva výber uzlov je zobrazený na obrázku 2.4.



Obr. 2.4: Minimax algoritmus[15]

- **Alfa-beta orezávanie** je vylepšenie Minimax algoritmu. Optimalizuje chod algoritmu tým, že vynecháva (orezáva) výpočty takých variácií, pri ktorých je v konkrétnom momente jasné, že nebudú konkurencieschopné k už vypočítaným variáciám. Príklad orezávania variácií je znázornený na obrázku 2.5. Podrobnejšie bude algoritmus opísaný v sekcii 2.4.1 v súvislosti so simulátorom Stockfish.



Obr. 2.5: Alfa-beta orezávanie[16]

2.4 Existujúce riešenia a ich princípy

V tejto sekcii budú analyzované existujúce šachové simulátory. Pri každom simulátore bude jeho stručná charakteristika, zasadenie do časového obdobia a aj vysvetlenie princípov jeho fungovania. Rovnako bude kapitola zameraná na vysvetlenie toho, aké sú medzi nimi rozdiely, a v poslednej časti budú porovnané ich súčasné výsledky a umiestnenia z hľadiska ELO hodnotenia.

2.4.1 Stockfish

Stockfish patrí medzi najznámejšie, a v súčasnosti najpoužívanejšie šachové simulátory. Patrí medzi najsilnejšie konvenčné šachové simulátory na svete.[17] Pojmom konvenčný simulátor sa budú označovať také šachové simulátory, ktoré majú spoločné nižšie uvedené znaky. Ich pôvod siaha do osemdesiatých rokov dvadsiateho storočia. Sú charakterizované tým, že:

- vytvárajú abstrakciu šachovej pozície,
- nachádzajú kandidátne ťahy,
- iterujú stromom kandidátnych ťahov do danej hĺbky,
- skúmajú vhodnosť týchto ťahov a hľadajú taký, ktorý je najviac vyhovujúci.

Jeho prvé vydanie bolo v roku 2008. V súčasnosti je voľne distribuovaný pod licenciou typu otvorený zdroj.

Abstrakcia šachovnice

Metodológia, na ktorej je založený Stockfish začína prevedením šachovnice na bitové pole. Keďže je šachovnica tvorená zo 64 štvorcov, vzniká z toho 64-bitová premenná. Každý bit reprezentuje jeden štvorec, a ak je nastavený na hodnotu 1, v tom prípade je na pozícii figúra. Bitové pole pešiakov tak môže byť reprezentované nasledovne (obrázok 2.6):

```
bitové pole:
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0

binárna hodnota:
00000000 00000000 00000000 00000000 00000000 00000000 11111111 00000000

decimálna hodnota:
65280
```

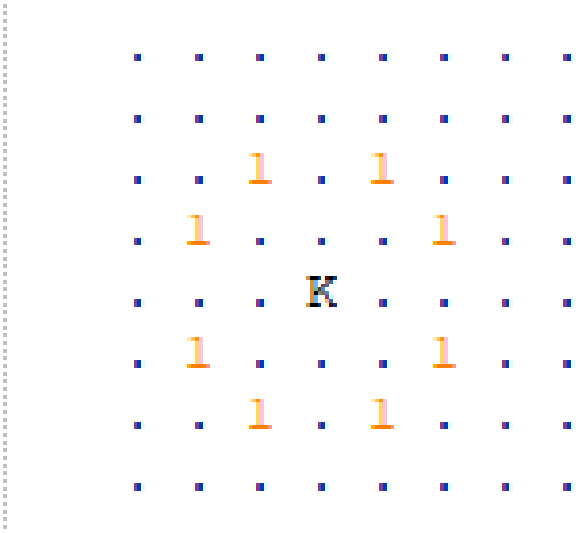
Obr. 2.6: Bitové pole využívané softvérom Stockfish

Takáto reprezentácia následne umožňuje bitové operácie. Napríklad, pohyb figúry o jednu pozíciu doľava je reprezentovaný ako 1-bitový posun doľava. Pohyb hore je reprezentovaný ako 8-bitový posun doľava. Získanie všetkých figúr hráča je logická operácia OR nad jednotlivými bitovými poľami pre každý typ figúry a pod.[18]

Generovanie ťahov

Následne Stockfish pokračuje generovaním ťahov. Podobne pri tom využíva bitové polia, kedy ma každý druh figúry presne definované pozície, ktoré môže z aktuálneho miesta dosiahnuť. Na obrázku 2.7 sú označené hodnotou 1 pozície, na ktoré môže útočiť jazdec (označený písmenom K, z angl. knight).

Množinu týchto potenciálnych ťahov je pomocou bitového maskovania (AND[19] operácia) možné prekryť množinou vlastných figúr. Týmto procesom je možné vylúčiť také pozície, na ktorých sa už nachádza figúra rovnakej farby - to nám pomáha k získaniu množiny možných (vykonateľných) ťahov. Pomocou binárnych operácií je teda možné definovať, na ktoré štvorce figúra smie ísť, či je práve pod útokom, alebo či chráni útok na kráľa (je teda zablokovaná).



Obr. 2.7: Možné ťahy[18]

Vyhľadávanie

Po generovaní ťahov nastáva vyhľadávanie najlepšieho z kandidátnych ťahov. Stockfish vyhľadáva pomocou Alfa-Beta procedúry (angl. Alfa-Beta pruning search). Alfa-beta procedúra je vylepšením Minimax vyhľadávania tým, že vynecháva variácie, ktoré by v optimálnej hre nikdy neboli dosiahnuté, pretože každý z hráčov vo svojom ťahu mení dej hry vo svoj prospech.

Keďže je (vo väčšine prípadov) výpočtovo neuskutočniteľné vyhľadávanie do konca hry, toto vyhľadávanie je ukončené po tom, čo dosiahne špecifikovanú hĺbku. Hĺbka vyhľadávania je meraná vo vrstvách, kde každá vrstva je ťah vykonaný hráčom. Vyhľadávacia hĺbka s hodnotou D indikuje, že vzdialenosť medzi koreňovým uzlom a (koncovým) uzlom vetvy je D vrstiev. Stockfish inkrementálne zvyšuje hĺbku vyhľadávacieho stromu, v procese nazývanom iteratívne hĺbenie. Avšak, dosiahnutie nominálnej vyhľadávacej hĺbky šachovým simulátorom neznamená, že zvážil všetky možné variácie D-počtu ťahov. To sa deje vďaka heuristike, ktorá smeruje simulátor k vyhľadávaniu sľubných variácií do väčšej hĺbky, a menej sľubných variácií do menšej hĺbky.[20] Príklad procedúry Alfa-Beta je zobrazený na obrázku 2.5.

Evaluácia

Evaluácia pozície je určenie heuristiky pre konkrétnu pozíciu. Nastáva vtedy, keď vyhľadávací algoritmus dosiahne koncovú vetvu stromu, a determinuje, pre ktorého hráča je pozícia výhodnejšia. Stockfish na evaluáciu používa pevne dané šachové koncepty, ako napríklad:

- **materiál** - hodnotí každý typ figúry inou hodnotou, poukazuje na nepomer medzi súpermi, keďže spravidla je pre hráča výhodnejšie mať viac materiálu,
- **pozícia figúr** - Stockfish má staticky implementovanú bitovú mapu, kde odlišuje preferované pozície figúry. Preferuje napríklad umiestnenie koňa do centrálnych štvorcov,
- **aktivita figúr** - či sú blokované, alebo sú naopak pohyblivé a majú na výber viacero ťahov,
- **fáza hry** - otvorenie, stredná hra, koncová hra.

2.4.2 AlphaZero

AlphaZero je šachový simulátor od spoločnosti DeepMind založený na strojovom učení. Dostal sa do popredia v roku 2017 po tom, čo jednoznačne porazil v tom čase úradujúci Stockfish v zápase ktorý pozostával zo 100 hier.[21] Jeho implementácia nie je verejne dostupná, avšak bolo zverejnených niekoľko odborných publikácií opisujúcich práve metodológiu tohoto simulátora. To viedlo k vývoju niekoľkých verejne dostupných projektov. Existuje teda niekoľko adaptácií neurónovej siete založených práve na AlphaZero, ako je napríklad Leela Chess Zero, Leelenstein, Alliestein atd.

Abstrakcia šachovnice

AlphaZero vytvára abstrakciu šachovnice tak, aby bola vhodná ako vstup do konvolučnej neurónovej siete. Podobne ako pri abstrakcii šachovnici, aj tu sa pracuje s binárnymi hodnotami. Keďže sa na dátach bude vykonávať konvolúcia, vstupom do neurónovej siete sú plochy o veľkosti 8x8, kde každý prvok kóduje typ figúry, ak sa nejaká na pozícii nachádza.

Generovanie ťahov

Možné ťahy, ktoré sa generujú podobne ako pri simulátore Stockfish sa využívajú spravidla po tom, čo neurónová sieť urobí predikciu. Na výstupný vektor je aplikovaná bitová maska tvorená z možných ťahov, čím sa zabezpečí, že výstup z neurónovej siete bude mať iba legálne hodnoty.

Vyhľadávanie

AlphaZero využíva algoritmus Monte Carlo Tree Search (MCTS) na identifikovanie najlepších ťahov za pomoci opakovaného vzorkovania (angl. repeated sampling). V MTCS dochádza k evaluácii na koncovom uzle simulácie.[22] To napomáha k výberu variácií, ktoré sú najviac sľubné. Hĺbka vyhľadávania sa zvyšuje každou simuláciou. Po niekoľkých simuláciách sa vyberie dcérsky uzol s najväčším množstvom vzoriek. Hodnota uzla je optimisticky odhadnutá pomocou Polynomial Upper Confidence Tree (PUCT) algoritmu.

Hodnota akcie (angl. action value) $Q(s, a)$ a vektor politík (angl. policy vector) $P(s, a)$ sú určené aplikáciou evaluačnej funkcie na stav v hernom strome. Vyhľadávací algoritmus je spočiatku zameraný na uzly s vysokou hodnotou akcie Q, a predchádzajúcou pravdepodobnosťou P. Podľa toho ako sa počet navštívení akcie a $N(s, a)$ zvyšuje, tak sa zvyšuje aj dôveryhodnosť algoritmu voči vzorkovanej hodnote Q.[20]

Evaluácia

AlphaZero využíva na evaluáciu hlbokú konvolučnú neurónovú sieť, ktorá bola už načrtnutá vyššie. Jej úlohou je predikovať vektor politík p_t a hodnotu v_t uzla vo vyhľadávacom strome. Dáta sú generované pomocou miliónov hier algoritmu hraného samého so sebou. Daným finálnym výstupom hry z (-1 pre prehru, 0 pre remízu a 1 pre výhru) a vektorom vyhľadaných pravdepodobností π_t získaných z koncového uzla, je sieť trénovaná na minimalizáciu stratovej funkcie pomocou vzorca:

$$(z - v)^2 - \pi^T \log p + c \parallel \theta \parallel^2$$

Kým hodnota v má priamo predikovať výsledok hry, p je trénovaná na predpovedanie ďalšieho vyhľadávania. Tento proces je snaha o naučenie sa užitočných predchádzajúcich pravdepodobností pre PUCT algoritmus.[20]

2.4.3 Porovnanie ELO hodnotení

Podľa portálu klubu CCRL (Computer Chess Rating Lists)[17], ktorý sa už od roku 2005 zaoberá ratingovým zoznamom šachových simulátorov, je v kategórii 40/15 (potreba vykonať 40 ťahov za 15 minút) na 1. mieste Stockfish 14. Ide o najnovšiu verziu softvéru Stockfish, ktorá má v ELO hodnotení 3542 bodov.

Ako už bolo spomínané v podkapitole o AlphaZero, jeho implementácia nie je prístupná verejnosti, takže jeho súčasnú výkonnosť nebolo možné preveriť. V

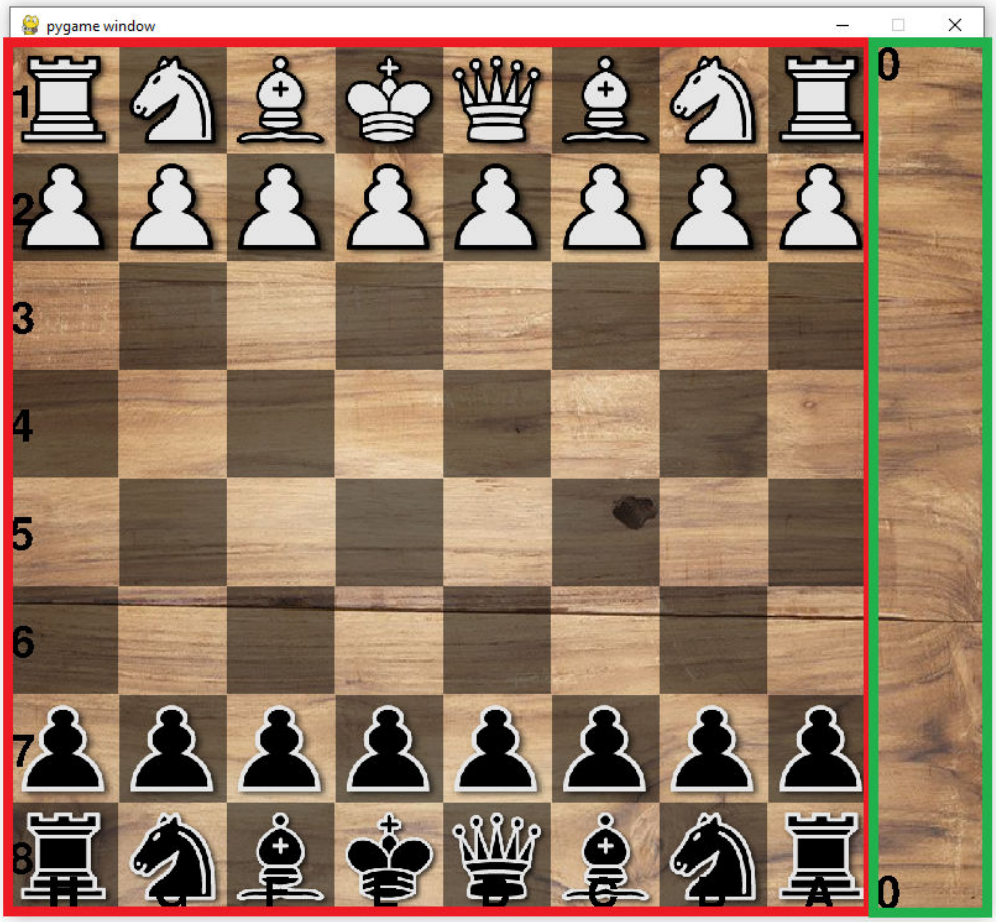
tabuľke však môžeme vidieť na 15. mieste Leela Chess verzie 0.28.0. Tento simulátor bol práve založený na princípoch AlphaZero, a je neformálne považovaný za jeho nasledovníka. Dosiahnuté skóre bolo 3378 ELO bodov.

V kategórii BLITZ (rýchly časový formát pozostávajúci z 2 minút na hráča a 1 sekundového inkrementu po ťahu) je taktiež na 1. Stockfish s hodnotením 3743. Leela Chess sa umiestnil na 4. mieste s 3669 bodmi. Uvedené údaje sú platné k dátumu 18.3.2022.

Napriek rozdielu 164 a 74 bodov ide v oboch prípadoch o veľmi precízne simulátory. Pre porovnanie, súčasný najlepší šachový veľmajster má 2865 ELO bodov[23], a hranica 2900 bodov nebola človekom nikdy prekonaná.

3 Tvorba používateľského rozhrania

Pre implementáciu bolo potrebné nájsť správny balík na tvorbu používateľského prostredia v jazyku Python. Ako najvhodnejší sa javí balík `pygame`[24], ktorý má všetku potrebnú funkcionalitu na tvorbu okna, grafických elementov alebo interakciu s prostredím. Ide o súbor niekoľkých modulov dizajnovaných na tvorbu videohier. Je dostupný zadarmo pod licenciou LGPL, čo umožňuje tvorbu komerčných, alebo verejne dostupných produktov. Ponúka podporu pre množstvo operačných systémov, a je možné ním programovať aj pomocou malého množstva kódu.



Obr. 3.1: Grafické rozloženie okna

3.1 Inicializácia

Ako prvé bolo potrebné vytvoriť grafické okno. Okno bude graficky zložené zo šachovnice s rozmermi 800x800px (červená farba na obr. 3.1), a bočného panelu s rozmermi 100x800px (zelená farba), v ktorom budú dodatočné informácie o šachovnici, ako napríklad rozdiel v materiáli a zoznam vyhodených figúr. Rozlíšenie o veľkosti 800x800 bolo určené z toho dôvodu, že ide o dostatočnú veľkosť pre šachovnicu. Okno tak bude viditeľné a nebude príliš veľké na menších obrazovkách, no je možné túto veľkosť kedykoľvek prispôsobiť. Do grafického okna, ktoré samé o sebe neobsahuje grafické prvky, bolo potrebné vložiť plochu typu `pygame.Surface`. Táto plocha bola predvolená na bielu farbu, a jej veľkosť je rovná veľkosti okna.

3.2 Cyklus vykreslenia

Ďalším krokom bolo vytvorenie cyklu, ktorý slúži na vykresľovanie obsahu okna po každom časovom intervale. Ide o takzvaný nekonečný cyklus, keďže vykresľovanie bude potrebné počas celého behu programu. Interval, ktorý je medzi dvomi vykresleniami sa vypočítava ako

$$I(s) = \frac{1(s)}{FPS}$$

kde hodnota FPS je konštanta predstavujúca počet snímok za sekundu. Pre potrebu práce bola používaná hodnota 30 snímkov za sekundu.

Pri každej iterácii je pripraviť jednotlivé elementy súvisiace s herným stavom, a následne zavolať metódu `pygame.display.update()`, ktorá daný obsah aktualizuje. Každé vykreslenie tak tvoria nasledujúce časti.

Vykreslenie pozadia

Ide o nastavenie celej plochy (pozadia) na bielu farbu. Alternatíva ku tomuto kroku je nahranie obrázka, ktoré bude slúžiť ako pozadie. Pre naše potreby bol použitý obrázok dreva pre dosiahnutie autentickejšieho vizuálneho zážitku pre používateľa.

Vykreslenie štvorcov

Je tvorené dvojrozmerným cyklom o veľkosti 8x8, kde každý štvorec je reprezentovaný malou plochou typu `pygame.Surface`. V každom nepárnom riadku je potrebné vyfarbiť čiernou farbou každý párny štvorec, v každom párnom riadku je

potrebné vyfarbiť čiernou farbou každý nepárny štvorec. Obsah štvorca je spravidla jedna šesťdesiatštvrtina obsahu celej šachovnice. V tomto prípade je rozmer štvorca 10x10. Keďže bude potrebné uložiť na štvorec aj figúry, rýdzo čierne štvorce by ich mohli prekrývať. Preto bola na štvorce aplikovaná hodnota `alpha=128`, oslabí vplyv čiernej farby a zabráni splynutiu.

Vykreslenie figúr

Prebieha na príslušnom štvorci podľa pozície figúry v hernom stave. Každá figúra má v repozitári umiestnenú príslušnú ikonu, ktorá ju bude na šachovnici reprezentovať. Načítanie ikony je vykonané pomocou príkazu `pygame.image.load()`. Funkcia vyžaduje jeden argument, a tou je cesta k obrázku alebo ikone. Tvorba cesty je v kóde tvorená genericky, a pozostáva z:

1. cesty k priečinku `images\sprites4`,
2. farby danej figúry `chess.COLOR_NAMES[piece.color]`,
3. typu danej figúry `chess.COLOR_NAMES[piece.color]`,
4. a prípony `.png`.

V prípade, že bude potrebné šachovnicu preklopiť (pre získanie pohľadu čierneho hráča), je potrebné pomocou funkcie `pygame.transform.flip()` obrázok taktiež preklopiť. Následne je obrázok škálovaný podľa príslušnej mierky ku veľkosti štvorca v ktorom sa nachádza, a pridaný do plochy. Použité ikony sú verejne dostupné na internetovej stránke *opengameart.org*[25] pod licenciou CC-BY-SA 3.0.

Vykreslenie odstránených figúr

Ide o Vykreslenie takých figúr, ktoré už boli vyhodené z hracej plochy. Tieto figúry sú načítané rovnako, ako bolo vysvetlené v predchádzajúcom kroku. Sú zobrazené v bočnom paneli, stále na príslušnej strane, ku ktorej patria. Každá ďalšia figúra, ktorá bola odstránená z hracej plochy vykreslená pod predchádzajúcu atd. Demonštráciu zoznamu odstránených figúr je možné vidieť na obrázku 3.2 v bočnom paneli napravo.

Obr. 3.2: Zoznam odstránených figúr v koncovke

Rozdiel materiálu

Zobrazenie rozdielu v materiáli medzi hráčmi. Keďže po každom ťahu je vypočítaná hodnota materiálu pre bieleho aj čierneho hráča, je možné pomocou rozdielu zistiť, ktorý hráč disponuje väčšou hodnotou materiálu. Rozdiel hodnôt je zobrazený obom hráčom na ich strane bočného panela, viditeľný na obrázku 3.2.

Otočenie obrazovky

Základný pohľad na šachovnicu je reprezentovaný z pohľadu bieleho hráča. Implementovaný bol však aj pohľad čierneho hráča, ktorý nastáva ak je `boolean` premenná `reverted` nastavená na pravdivú hodnotu. V tomto prípade je potrebné po vykreslení predchádzajúcich elementov otočiť grafické okno príkazom `screen.blit(pygame.transform.rotate(screen, 180), (0, 0))`.

Vykreslenie nápovedy k súradniciam

Ku šachovnici štandardne patrí aj informácia o súradniciach po stranách šachovnice, ktorá slúži na lepšiu orientáciu a značenie štvorcov. Stĺpce šachovnice sa zľava doprava zo strany bieleho označujú písmenami abecedy od 'A' po 'H'. Rady šachovnice sa označujú zdola nahor číslicami od '1' po '8'. Pre pohľad čierneho hráča je potrebné tieto informácie otočiť, teda rady označovať číslami zhora nadol, a písmená vypísať sprava doľava.

3.3 Udalosti

Udalosti slúžia na zachytávanie interakcie od používateľa, a umožňujú programu na takéto interakcie adekvátne reagovať. Udalosti sa získavajú pri každej iterácii pomocou príkazu `pygame.event.get()`, avšak kvôli prehľadnosti sú opísané v tejto samostatnej podkapitole. Zoznam zachytávaných udalostí je nasledovný:

- Stlačenie klávesy `pygame.K_RIGHT` - naviguje na nasledujúci ťah,
- stlačenie klávesy `pygame.K_LEFT` - naviguje na predchádzajúci ťah,
- stlačenie klávesy `pygame.K_s` - zobrazí SVG obrázok (obr. 1.1) s aktuálnou pozíciou
- stlačenie klávesy `pygame.K_r` - vertikálne otočí zobrazenie
- stlačenie tlačidla myši `pygame.MOUSEBUTTONDOWN` - označí alebo zoberie figúru, ktorou sa bude hýbať,
- pustenie tlačidla myši `pygame.MOUSEBUTTONUP` - pustí alebo umiestni označenú figúru na danú pozíciu.

Pri kliknutí myšou je stále potrebné vypočítať, nad akým štvorcom k udalosti došlo. Pomocou funkcie `pygame.mouse.get_pos()` je možné zistiť presnú hodnotu `x` a `y` pixelu myši v čase, keď sa táto udalosť vykonala. Následne je potrebné iterovať všetkými štvorcami a zistiť, ktorý štvorec súvisí s pozíciou. To je zobrazené v zdrojovom kóde 3.1.

```
def get_piece_from_mouse_position(pos, squares):
    for row in squares:
        for s in row:
            if s.get_x_start() <= pos.x and
            s.get_x_end() > pos.x and
            s.get_y_start() <= pos.y and
            s.get_y_end() > pos.y:
                return s.piece
    return None
```

Zdrojový kód 3.1: Získanie figúry podľa pozície myši

4 Predspracovanie dát

Dáta určené pre imitačné učenie (popísané neskôr v kapitole 6) boli získané z Lichess Elite databázy[26]. Databáza obsahuje šachové hry hrané používateľmi, ale aj umelou inteligenciou na portáli *lichess.org*. Cieľom databázy je filtrovať všetky hry z tohoto portálu tak, aby zostali iba také, kde jeden z hráčov má ELO hodnotenie nad 2400, a druhý hráč nad 2200 ELO bodov. Vylúčené boli tiež rýchle (angl. bullet) hry. Všetky hry sú štandardného formátu (nejde teda o nejaký šachový variant). Pre účely tejto práce sa pracuje s dátami pozostávajúcimi z 1 995 679 šachových hier hraných v mesiacoch november a december 2021, a január 2022. Dôvod, prečo boli vybrané tieto dáta je, že ide o moderné hry hrané v súčasnosti. Vďaka kvantite dát bude zabezpečená ich odlišnosť, a obsiahnutie všetkých štandardných otvorení. Ich celková veľkosť na disku je 1,68GB, a sú v štandardnom formáte PGN (Portable Game Notation).

PGN súbor je najprv potrebné načítať do pamäte. To je vykonané pomocou funkcie `open()` z python balíka `io`, ktorá otvorí súbor, a následne funkcie `read()` nad súborom, pre načítanie dát. Tieto načítané dáta sú vo forme objemného textového reťazca, ktorý obsahuje všetky hry. Je preto potrebné rozdeliť text na jednotlivé hry funkciou `split()`, čím sa vytvorí pole reťazcov reprezentujúcich hru. Počet hier je možné dynamicky vypočítať podľa vstupného PGN súboru funkciou `len()`, a uložiť do premennej.

Pre prácu s jednotlivými hrami ich bolo potrebné previesť z reťazcov na PGN objekty. To je možné za pomoci volania `chess.pgn.read_game()` z balíka `chess`. Z PGN objektu bolo potrebné získať šachovnicu, a následne získať objekty ťahov volaním metódy `mainline_moves()` nad PGN objektom. Všetky tieto ťahy bolo potrebné iterovať, a postupne ich vykonávať vložením do objektu šachovnice. Pri každom takomto kroku sa po vložení ťahu vytvorila jedna vzorka, ktorá je vhodne naformátovaná pre neurónovú sieť. Každá vzorka je tvorená z dvoch častí, ktoré sú popísané v nasledujúcich sekciách.

4.1 Primitívna reprezentácia šachovnice

Zjednodušená reprezentácia šachovnice je vytvorená z objektu šachovnice. Pozostáva z trojrozmerného poľa 8x8x12, kde 8 reprezentuje počet stĺpcov, 8 reprezentuje počet riadkov, a 12 reprezentuje typ figúry v binárnom poli. Táto reprezentácia šachovnice bude vstupom do neurónovej siete. Jej tvorba je zobrazená v nasledujúcom zdrojovom kóde 4.1.

```
def create_raw_board(myBoard: MyBoard):
    arr = [
            [[0 for col in range(12)] for row in range(8)]
            for x in range(8)
          ]

    whiteShift, blackShift = -1, 5
    for i in range(64):
        piece = myBoard.board.piece_at(i)
        if(piece is not None):
            sqX = chess.square_file(i)
            sqY = chess.square_rank(i)
            if(piece.color):
                arr[sqX][sqY][piece.piece_type+whiteShift]=1
            else:
                arr[sqX][sqY][piece.piece_type+blackShift]=1

    return np.asarray(arr)
```

Zdrojový kód 4.1: Tvorba primitívnej reprezentácie šachovnice

4.2 Pole akcií

Pole akcií je jednorozmerné binárne pole o dĺžke 4096. Slúži na kódovanie ťahov, aby boli vhodné ako výstup z neurónovej siete. Každý element predstavuje potenciálny ťah, keďže je možné sa s figúrou pohnúť zo 64 možných štvorcov na jeden zo cieľových 64 štvorcov (v praxi je možné premiestniť figúru na 63 štvorcov, keďže ťah na rovnaký štvorec, na ktorom sa figúra nachádza nie je možný, avšak ide o zjednodušenie). Tvorba poľa je zobrazená aj s vysvetľujúcimi komentármi v zdrojovom kóde 4.2.

```
# vytvorenie pomocného 1D poľa s hodnotami od 0 po 4096
allMoveIndexes = np.arange(64 * 64)

# transformácia 1D poľa na 2D pole
allMoveIndexes = allMoveIndexes.reshape(64, 64)

# získanie indexu, kde sa konkrétny ťah nachádza, za pomoci
# začiatočného a koncového štvorca ťahu
idx = allMoveIndexes[move.from_square, move.to_square]

# inicializácia poľa ťahov o dĺžke 4096 a nastavenie jeho hodnôt na nuly
allMoves = np.zeros(64 * 64, dtype=int)

# nastavenie očakávaného ťahu na hodnotu 1
allMoves[idx] = 1
```

Zdrojový kód 4.2: Tvorba poľa akcií

5 Algoritmus na princípoch AlphaZero

Keďže pri analýze existujúcich riešení bolo zistené, že algoritmy založené strojovom učení majú tendenciu prekonávať štandardné implementácie (založené na pevne naprogramovaných pravidlách), prvotným pokusom o implementáciu umelej inteligencie v šachu bude algoritmus založený na princípe učenia s posilňovaním AlphaZero. Ako už bolo spomínané v analytickej časti v sekcii 2.4.2, pôvodná implementácia AlphaZero nie je verejne dostupná. Existuje však verejne dostupná práca Naira[27] zverejnená pod Stanfordskou univerzitou, ktorá je založená na pôvodných dokumentoch AlphaGo Zero od spoločnosti DeepMind. Tieto dokumenty sa zaoberali strojovým učením hry Go, avšak ich princípy sú aplikovateľné aj na iné typy hier.

Súčasťou spomínanej práce je aj zverejnený git repozitár[28], ktorý obsahuje existujúcu implementáciu strojového učenia pre hru Othello. Repozitár je možné voľne rozširovať o iné typy hier, čo bude slúžiť práve na implementáciu hry šach. Cieľom tejto kapitoly tak bude implementovať a reprodukovať správanie AlphaZero pomocou spomínaného repozitára, a zhrnúť dosiahnuté výsledky. Zdrojové kódy pochádzajúce z repozitára, ktoré budú využité a modifikované v rámci tejto práce budú riadnym spôsobom odkazovať na pôvodný zdroj uvedený v komentári v konkrétnom súbore.

Strojové učenie na základoch AlphaZero je založené na tom, že agent hraje hry sám so sebou. Z poznatkov, ktoré počas hier získa sa agent učí. Nepotrebuje k tomu poznať žiadne dáta alebo znalosť danú človekom, iba základné pravidlá danej hry. Kombinuje neurónovú sieť, a vyhľadávací algoritmus Monte Carlo v iteratívnom učení. Vyhľadávaniu Monte Carlo bola venovaná pozornosť v sekcii 2.4.2, v časti "Vyhľadávanie".

5.1 Architektúra neurónovej siete

Jedným z hlavných prvkov riešenia je neurónová sieť. Keďže pozostáva z viacerých skrytých vrstiev, hovoríme, že ide o hlbokú neurónovú sieť. Ako uvádza aj

Nair vo svojej práci, sieť f_θ je parametrizovaná hodnotou θ a berie ako vstup momentálny stav s na šachovnici. Model má dva výstupy. Prvým je spojitá hodnota stavu na šachovnici $v_\theta(s)$ nadobúdajúc hodnoty v intervale od -1 po 1 z pohľadu konkrétneho hráča, ktorý je v stave s na ťahu. Druhým je vektor politík $\vec{p}_\theta(s)$, ktorý je vo svojej podstate poľom pravdepodobnostných hodnôt pre každú akciu. Spôsob, akým sa tieto vstupné a výstupné hodnoty tvoria (kódujú) je popísaný v sekciách 4.1 a 4.2.

Strojové učenie je založené na python knižnici `torch`. Samotná neurónová sieť je vytvorená ako trieda v jazyku python, ktorá dedí od triedy `torch.nn.Module`. Obsahuje dve metódy. Konštruktor, ktorý nastaví hodnoty dimenzií, s ktorými sa bude pracovať, nastaví dĺžku výstupu, a potrebné hyperparametre potrebné pre trénovanie. Použité hyperparametre sú zobrazené v zdrojovom kóde 5.1.

```
args = dotdict({
    'lr': 0.001,
    'dropout': 0.3,
    'epochs': 10,
    'batch_size': 64,
    'cuda': torch.cuda.is_available(),
    'num_channels': 512,
})
```

Zdrojový kód 5.1: Hyperparametre použité pri učení AlphaZero

Model pozostáva zo štyroch konvolučných vrstiev. Všetky konvolučné vrstvy majú filter o veľkosti 3x3 a používajú hodnotu `stride`=1, teda posun filtra prebieha stále o jednu pozíciu. Prvé dve vrstvy využívajú `padding`=1, ktorý zachová veľkosť výstupných dimenzií. Druhé dva vrstvy `padding` nevyužívajú, čo následne zníži počet výstupných dimenzií. Ak bude vstupom do siete herný stav o dimenzii 8x8 (v prípade šachu), po aplikácii týchto štyroch vrstiev sa tak zníži počet dimenzií na 4x4. Model ďalej obsahuje štyri 2d Batch normalizačné vrstvy, štyri lineárne vrstvy, a dve 1d Batch normalizačné vrstvy.

Druhou metódou je metóda `forward(self, s)`, ktorá slúži na prácu s jednotlivými vrstvami. Jej vstupom je herný stav, ktorý postupne prechádza cez všetky vrstvy neurónovej siete, a vráti vektor politík $\vec{p}$, a hodnotu v. Na výsledok z každej zo štyroch konvolučných vrstiev (`conv1` - `conv4`) je aplikovaná ešte 2d Batch Normalizácia (`bn1` - `bn4`), a následne Rectified Linear Unit (ReLU) aktivačná funkcia. Následne sú pridané dve lineárne vrstvy (fc1, fc2) s 1d Batch normalizáciou, ReLU aktivačná funkcia a Dropout.

5.2 Implementácia šachovej logiky

Pre správnu integráciu hry šach s modelom je potrebné implementovať novú triedu `ChessGame`, ktorá bude dediť od existujúcej triedy `Game`, a prekryť jej metódy. Tieto metódy budú obsahovať logiku hry šach:

- **`__init__(self)`** - konštruktor triedy slúžiaci na inicializáciu potrebných členských premenných.
- **`getInitBoard(self)`** - vytvorí novú šachovnicu v základnom postavení príkazom `chess.Board()`, a vráti ju ako výstup.
- **`getBoardSize(self)`** - vráti premennú typu `tuple` obsahujúcu dimenzie šachovnice `(8,8,12)`.
- **`getActionSize(self)`** - vráti celkový počet možných akcií o hodnote `64*64`.
- **`getNextState(self, board, player, action)`** - metóda aplikuje na šachovnicu akciu prichádzajúcu v parametri príkazom `board.push(move)`, a vráti túto zmenenú šachovnicu súčasne s novým hráčom, ktorý je na ťahu.
- **`getValidMoves(self, board, player)`** - metóda podľa šachovnice zistí, aké sú v danej pozícii pre hráča dostupné ťahy, a vráti binárne pole akcií o dĺžke 4096, kde hodnota 0 bude na nedostupných akciách, a hodnota 1 bude na dostupných akciách.
- **`getGameEnded(self, board, player)`** - metóda zistí súborom podmienok (napr. `board.is_checkmate()`), či je pre hráča v danom stave hra výhra, prehra, remíza, alebo sa pokračuje v hre.
- **`getCanonicalForm(self, board, player)`** - vráti kanonickú formu hry. Ak ide o bieleho hráča, vráti aktuálnu verziu šachovnice. Ak ide o čierneho hráča, vráti vertikálne otočenú šachovnicu.
- **`getSymmetries(self, board, pi)`** - vytvorí symetrické transformácie šachovnice. Metóda slúži na zväčšenie objemu trénovacích vzoriek tým, že z jednej pozície v hre je možno urobiť 8 transformácií. Tento prístup nebol reálne v tejto šachovej implementácii využitý.
- **`stringRepresentation(self, board)`** - vráti textovú reprezentáciu danej šachovnice. Na jej tvorbu bol využitý príkaz `board.epd()`. Zmyslom tejto metódy je zabezpečiť, aby Monte Carlo algoritmus vedel vďaka textovému reťazcu rozlišovať, či pozícia už bola navštívená, alebo nie.

5.3 Priebeh učenia

Hlavnou myšlienkou procesu učenia je očakávanie, že sa model časom naučí, ktoré stavy budú nakoniec viesť k výhram alebo prehrám. Naviac, učenie politík bude dávať dobrý predpoklad, aká je najlepšia akcia z daného stavu. Na začiatku je inicializovaná nová neurónová sieť s náhodnými váhami, teda začínajúc s náhodnými politikami a hodnotou v. Algoritmus začína cyklom o počte 100 iterácií. Každá iterácia bude pozostávať z 100 epizód, teda 100 hier hraných so sebou. Pri každom ťahu v hre je vykonaných 150 MCTS simulácií začínajúcich z aktuálneho stavu s. Je vybraný najviac vyhovujúci ťah z aktuálneho vektora politík. Týmto procesom je získaná trénovacia vzorka. Po skončení hry je ku každej vzorke pridaná hodnota od -1 po 1, podľa výhry alebo prehry daného hráča. Na konci iterácie je neurónová sieť trénovaná pomocou získaných vzoriek. Stará a nová (trénovaná) sieť so sebou súperia v 20 hrách. Ak nová sieť vyhrá percentuálne viac hier, ako je 55 percentný limit, je táto sieť prijatá a bude vybraná do ďalšej iterácie. V prípade že tento limit nesplní, bude vybraná do ďalšej iterácie pôvodná sieť.

5.4 Overenie modelu

Napriek teoretickým zisteniam, ktoré ukazujú, že kombinácia hry siete samej so sebou a učenia s posilňovaním by mala priniesť výrazne lepšie výsledky v porovnaní s inými riešeniami, toto riešenie v praxi neprinieslo požadované výsledky. Pri behu algoritmu sa začal javiť problém s veľkou časovou náročnosťou pri hre šach, či už pri simulovaní hry algoritmom Monte Carlo, alebo pri samotnom trénovaní siete. Keďže pri každom ťahu hry je potrebné použiť vyhľadávanie MCTS, jedna epizóda trvala priemerne 10 minút, v závislosti od dĺžky hry. Z tohoto dôvodu by trvanie jednej iterácie trvalo viac ako 16 hodín, bez zahrnutia trénovania. Odhadom by bolo potrebné vykonať stovky takýchto iterácií, čo by znamenalo niekoľko mesačný chod programu. V prípade aj malej chyby v algoritme by musel byť proces prerušený, a spustený od začiatku, alebo od posledného platného modelu. Takýto prístup bol preto posúdený ako neoptimálny vzhľadom k časovému obmedzeniu.

Pokus o alternatívu bol vykonaný tým, že sa znížil počet epizód na 50, a počet MCTS simulácií zo 150 na hodnotu 50. To síce vyústilo k zlepšeniu časovej náročnosti, avšak pri porovnávaní starého a nového modelu došlo iba sporadicky k výhre nového modelu. Výsledky boli nedostatočné pravdepodobne preto, lebo

došlo k zníženiu počtu vzoriek na trénovanie a ich kvality. Z dôvodu nedostatočných výsledkov tak bola práca na tomto riešení uzatvorená. Potenciálnym riešením problému by v budúcnosti mohlo byť použitie výkonnejšieho hardvéru. Pravdepodobne by išlo o výkonnejší procesor od použitého procesora *Intel Core i5-9300HF 2.40GHz*, a zmeny sekvenčného Monte Carlo algoritmu na paralelný.

6 Algoritmus založený na imitačnom učení

Ďalším krokom bolo hľadanie spôsobu, akým nahradiť algoritmus založený na AlphaZero. Keďže sa generovanie ťahovou pomocou algoritmu MCTS ukázalo ako hlavný faktor zvyšujúci časovú komplexnosť, bolo potrebné sa zamerať na tento aspekt. Riešením, ako sa vyhnúť časovo náročnému získavaniu vzoriek, je úplné vylúčenie algoritmu Monte Carlo z procesu trénovania, a namiesto toho získanie vzoriek z už existujúcich hier. Algoritmus sa bude snažiť imitovať akcie z týchto hier, preto bude nazývaný imitačný, resp. imitačné učenie. Spôsob, akým boli vzorky nadobudnuté a spracované, je bližšie popísaný v kapitole 4. V riešení nastala aj principiálna zmena. Kým algoritmus AlphaZero využíval učenie s posilňovaním, v tomto riešení už pôjde o učenie s učiteľom.

6.1 Architektúra neurónovej siete

Architektúra neurónovej siete zostáva od predchádzajúcej kapitoly nezmenená, keďže sa nevyskytol konkrétny dôvod túto sieť meniť. Sieť stále pozostáva zo štyroch konvolučných vrstiev. Sieť však už nebude pracovať s hodnotou v, ostane však vektor politík $\vec{p}$, ktorý bude slúžiť na predikovanie ďalšej akcie pomocou neurónovej siete. Mierne zmenené boli aj hyperparametre. Parameter rýchlosti učenia `lr` bol znížený na hodnotu 0.00001. Dôvodom je, že sa bude pracovať s pomerne veľkou vzorkou dát. Pri vysokej hodnote `lr` by mohlo dochádzať k prílišnému naviazaniu siete ku čiastkovým dátam (spomínaných 2000 vzoriek), čo by znamenalo pretrénovanie, a zlú presnosť pri vložení nových vstupných dát do siete. Znížil sa tiež počet epoch na hodnotu 1, keďže nie je potrebné opakovane učiť model na čiastkové dáta, ale skôr kumulovať učenie pomocou práce zo stále novými vzorkami. Zmenené hyperparametre sú zobrazené v zdrojovom kóde 6.1.

```
args = dotdict({
    'lr': 0.00001,
    'dropout': 0.3,
    'epochs': 1,
    'batch_size': 64,
    'cuda': torch.cuda.is_available(),
    'num_channels': 512,
})
```

Zdrojový kód 6.1: Hyperparametre použité pri imitačnom učení

6.2 Priebeh učenia

Algoritmus začína načítaním dát zo súboru a ich spracovaním. Po zistení počtu načítaných hier (necelé dva milióny) je tento údaj vypísaný do štandardného výstupu pre overenie načítania. Následne sa určí počet vzoriek, ktoré budú zahrnuté v jednej iterácii programu. Pre tieto účely bola zvolená hodnota 2000, čo reprezentuje počet ťahov, ktoré budú v jednej chvíli vstupom do neurónovej siete. Vytvorí sa inštancia triedy `ChessGame`, ktorá obsahuje pomocné metódy pre prácu so šachovou logikou. Na znižovanie chybovosti neurónovej siete bol použitý optimalizátor Adam. Jeho vstupom boli parametre neurónovej siete, a spomínaný parameter `lr` definujúci rýchlosť učenia. Pre účely dokumentovania bol na vypísaný do konzoly čas začiatku učenia, a použité parametre.

Hlavnou časťou algoritmu je nekonečný cyklus. Dôvodov zvolenia takéhoto cyklu je viacero. Zvyčajne je na učenie použitý konečný cyklus o počte iterácií a epizód. V prípade že je trénovacia vzorka menšia, stačí jedno spustenie programu, a pri zvolení správnej neurónovej siete to stačí na naučenie daného modelu. Keďže je však celkový počet dát využitých v tejto práci pomerne rozsiahly, a hra šach je pomerne komplexná s vysokým faktorom vetvenia, bol zvolený práve nekonečný cyklus, ktorý berie stále náhodnú podmnožinu dát z celkovej vzorky. Ďalším dôvodom je praktickosť. Je postačujúce spustiť program raz, a pravidelne ukladať lepšie a lepšie naučený model do samostatných súboroch každý časový úsek. Algoritmus tak môže bežať aj niekoľko dní bez potreby zásahu, alebo výpočtu, koľko iterácií bude pravdepodobne potrebných. Je tak možné sa vyhnúť napríklad prípadu, kedy chybou programátora bude zvolený počet iterácií malý. V tomto prípade algoritmus skončí rýchlo, napríklad v noci, bez očakávaného výsledku. V prípade, že programátor nie je k dispozícii, výpočtový výkon zostáva

nevyužitý do príchodu programátora.

Cyklus začína výberom náhodnej hry z načítaného súboru a spracovaním jej ťahov na vytvorenie vzoriek. Keďže štandardne hry obsahujú približne 60 ťahov, teda 120 polťahov, je z jednej takejto hry vygenerovaných 120 vzoriek. Proces výberu náhodnej hry a jej spracovania sa opakuje dovtedy, kým nie je nahromadených 2000 vzoriek. Vzorky sú po spracovaní zoradené od začiatku hry do konca, kde každá ďalšia hra pridá do zoznamu svoje vzorky v takomto zoradení. Na to, aby sa model pri učení neviazal na takéto poradie dát, sú nahromadené vzorky premiešané funkciou `random.shuffle(trainExamples)`. V tomto momente je podmnožina vzoriek spracovaná a pripravená na trénovanie. Trénovanie prebieha opakovanou predikciou akcie podľa stavu, a následného určenia chybovosti chybovou funkciou. Celková chybovosť je spätne propagovaná neurónovou sieťou, a vďaka optimalizátoru Adam sú aktualizované váhy funkciou `optimizer.step()`. Ukážka cyklu je zobrazená v zdrojovom kóde 6.2.

```
while 1:
    trainExamples = []
    pbar = tqdm(total=samples_in_training)

    while len(trainExamples) < samples_in_training:
        randomPgnText = random.choice(text_pgns)
        curPgn = chess.pgn.read_game(io.StringIO(randomPgnText))
        newSamples = parse_game(curPgn)
        trainExamples += newSamples
        pbar.update(len(newSamples))
        totalGamesVisited += 1
    pbar.close()

    random.shuffle(trainExamples)
    nnet.train(trainExamples[:samples_in_training],
               externalOptimizer=optimizer)
```

Zdrojový kód 6.2: Cyklus využitý pri imitačnom učení

V pravidelných intervaloch je aktuálny model uložený do samostatného súboru s koncovkou `.tar`. Deje sa to za pomoci funkcie `torch.save()`, ktorá uloží slovník stavov neurónovej siete. V súčasnosti je zvolená hodnota 500, kedy sa pri každej 500. iterácii uloží model do súboru. Neskôr je možné pri predikcii takýto súbor načítať funkciou `torch.load()`, ktorá novému modelu priradí stavy zo sú-

boru.

Pri učení oboch hráčov naraz vznikla obava, že by model nemusel byť schopný posúdiť, ktorého hráča v konkrétnom stave uprednostniť. Kvôli jednoduchosti a jednoznačnosti učenia, a vzorkovania hier pre neurónovú sieť bolo učenie rozdelené ne trénovanie bieleho hráča, a čierneho hráča. Pri predikcii, alebo pri hre v používateľskom rozhraní tak bude potrebné načítať model správneho hráča.

6.2.1 Trénovanie modelu bieleho hráča

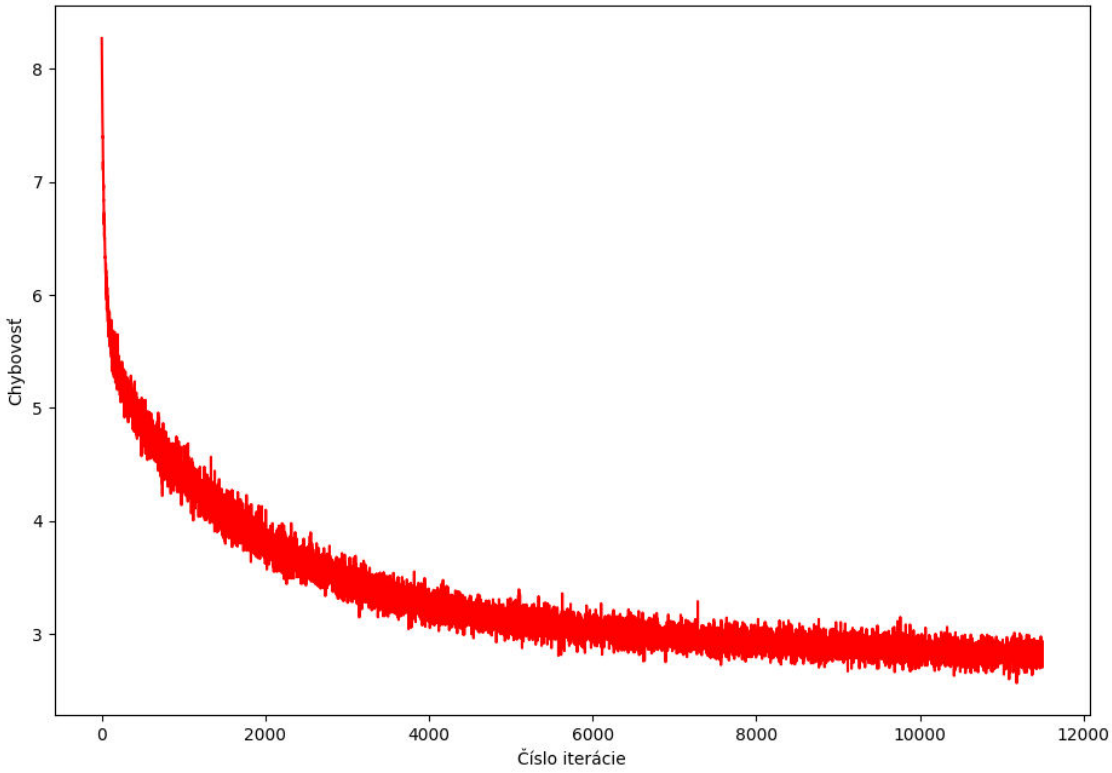
Trénovanie bieleho hráča pozostáva zo správneho rozdelenia vzoriek z hry pre vstup do neurónovej siete. Vzorky z hry sú na začiatku zoradené v poradí od začiatku hry do konca. Cieľom trénovania je dosahovať najlepšiu možnú presnosť ťahov. Z hry sa teda vyberajú také vzorky, ktoré sú vo výhernej hre bieleho. Keďže prvý ťah začína biely hráč, index tohoto ťahu sa rovná nule. Po ťahu čierneho hráča, ktorý má index jedna, pokračuje opäť biely hráč, s indexom ťahu 2. Z uvedeného vyplýva, že ak vyhrá biely hráč, je potrebné na trénovanie pripraviť všetky párne vzorky. Príklad výberu vzoriek je zobrazený v zdrojovom kóde 6.3.

```
out = []
if board.is_checkmate():
    # ak dal mat biely hráč, board.turn vráti False
    if not board.turn:
        out = currentExamples[::2]
else:
    out = currentExamples[::2]

return out
```

Zdrojový kód 6.3: Výber vzoriek bieleho hráča

Na obrázku 6.1 je zobrazený graf učenia bieleho modelu. Kým na y-osi je zobrazená chybovosť predikcie, x-os predstavuje počet vykonaných iterácií. Z grafu vyplýva, že chyba vypočítaná chybovou funkciou sa s narastajúcim počtom vykonaných iterácií znižuje. Kým v prvých iteráciách bola chyba rovnajúca sa hodnote približne 8,27, postupným trénovaním modelu sa kontinuálne hodnota znižovala až na hodnotu približne 2,70. Počet vykonaných iterácií bol 11500. Každá iterácia obsahovala spomínaných 2000 vzoriek. Pri tejto hodnote už výkonnosť trénovania klesala, preto bolo rozhodnuté o ukončení trénovania.



Obr. 6.1: Graf trénovania modelu bieleho hráča

6.2.2 Trénovanie modelu čierneho hráča

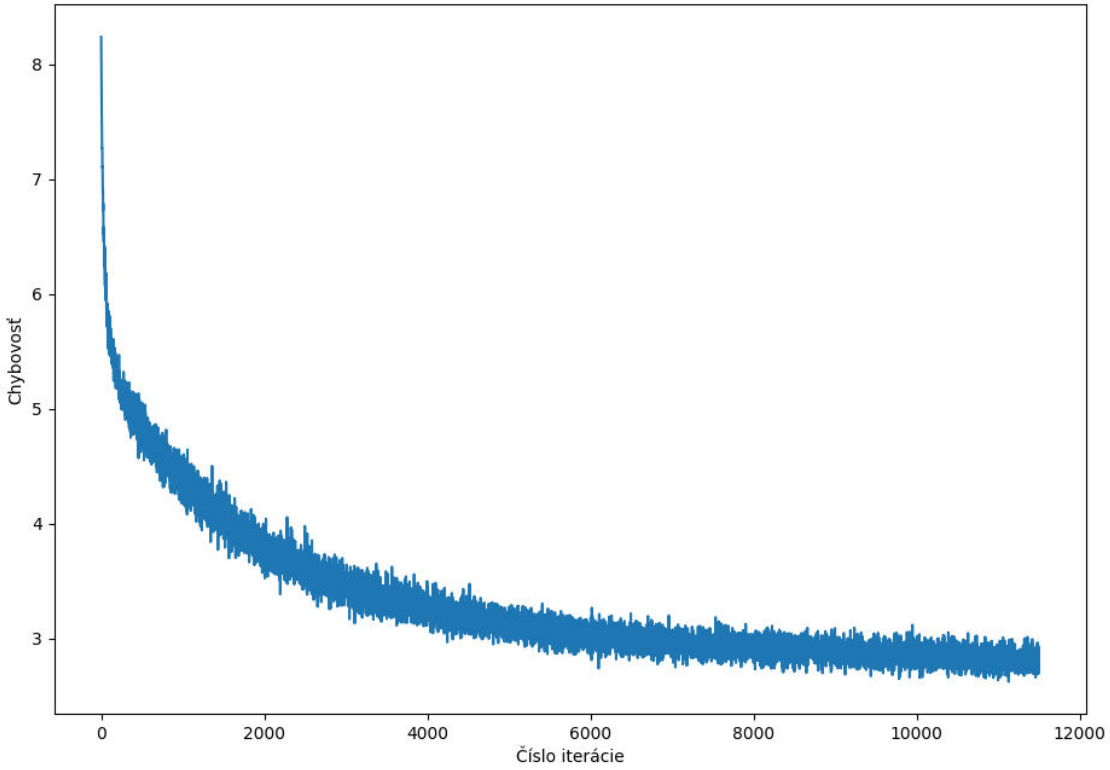
Trénovanie čierneho prebiehalo analogicky, pozostávajúc zo správneho rozdelenia vzoriek z hry pre vstup do neurónovej siete. Z hry boli vybrané také vzorky, ktoré sú vo výhernej hre čierneho. V tomto prípade bolo potrebné vybrať všetky nepárne vzorky. Príklad výberu vzoriek je zobrazený v zdrojovom kóde 6.4.

```
out = []
if board.is_checkmate():
    # ak dal mat čierny hráč, board.turn vráti True
    if not board.turn:
        out = currentExamples[1::2]
else:
    out = currentExamples[1::2]

return out
```

Zdrojový kód 6.4: Výber vzoriek čierneho hráča

Na obrázku 6.2 je zobrazený graf učenia čierneho modelu. Graf zobrazuje podobný typ krivky, aká bola pri učení čierneho modelu. Kým v prvých iteráciách bola chyba rovnajúca sa hodnote približne 8,24, postupným trénovaním modelu sa kontinuálne hodnota znižovala až na hodnotu približne 2,71. Počet vykonaných iterácií bol rovnako 11500, pozostávajúc z 2000 vzoriek. Pri tejto hodnote už výkonnosť trénovania klesala, preto bolo rozhodnuté o ukončení trénovania.



Obr. 6.2: Graf trénovania modelu čierneho hráča

6.3 Overenie modelu

Overenie modelu prebiehalo experimentálne. Proti modelu bolo odohraných niekoľko zápasov ľudským hráčom, rovnako ako prebiehalo porovnanie algoritmov pomocou vzájomných zápasov, ktoré bude popísané v kapitole 8. Pri týchto prípadových štúdiách bola následne aplikovaná analýza šachovej partie dostupná na portáli *chess.com*[29].

Výsledky všetkých zápasov, ktoré boli hrané modelom sa následne sprieme-rovali. Priemerná presnosť ťahov tvorených imitačným modelom bola 64%. Ide o dôstojný výsledok vzhľadom na fakt, že model generoval ťahy bez akejkoľvek znalosti pevne daných šachových princípov. Model vykazoval dobrú znalosť v imitovaní pozícií, s ktorými už mal skúsenosť pri trénovaní, a to hlavne v otvo-

rení. Model je teda vhodný pre hľadanie pozičných ťahov, napríklad rozvíjanie pozícií figúr zo štartovacej pozície, alebo obsadenie stredu na šachovnici.

Na druhú stranu, model zaostával v niektorých taktických pozíciách, hlavne v neskoršej fáze hry. Tie vyústili v stratu materiálu, prípadne prehru. Išlo o pozície, ktoré model pri trénovaní nemal k dispozícii. Napriek tomu, že pri trénovaní bola k dispozícii pomerne veľká vzorka hier, kvôli komplexnosti šachu existuje veľké množstvo pozícií, a nie je možné zachytiť pri trénovaní každý jeden prípad. Ďalším dôvodom slabších výsledkov v niektorých pozíciach je fakt, že tento model nevyužíva žiadne dopredné vyhľadávanie. Bez informácie, ako bude situácia vyzerať o niekoľko ťahov dopredu tak model nie je schopný predpovedať stratu materiálu. Ďalej budú pokračovať štúdie konkrétnych prípadov.

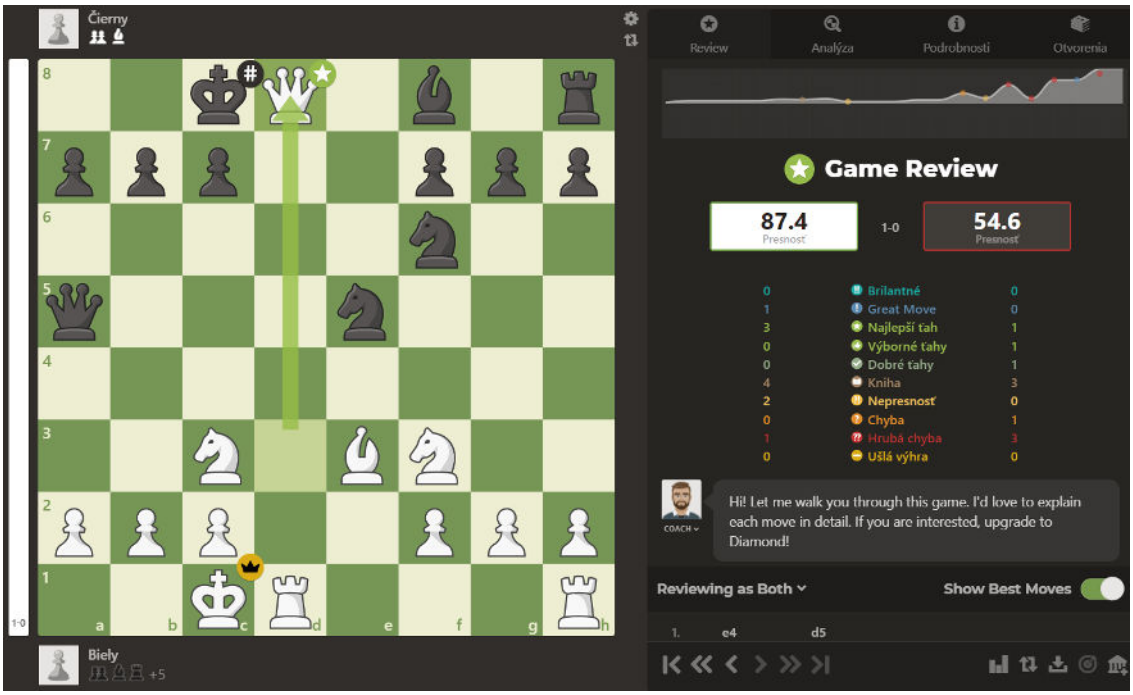
V nasledujúcej hre je vo formáte PGN zobrazený zápas medzi modelom (biely), a ľudským hráčom. Zápas vyhral model s presnosťou 87,4%:

```
[Site "Chess.com"]
[Date "2022.04.10"]
[White "Model"]
[Black "Človek"]
[Result "1-0"]

1. e4 d5 2. exd5 Qxd5 3. Nc3 Qa5 4. d4 Bf5 5. Bd3 Bxd3 6. Qxd3 Nd7 7.
Nf3 O-O-O 8. Be3 Ngf6 9. O-O-O e5 10. dxe5 Nxe5 11. Qxd8# 1-0
```

Obr. 6.3: Napodobenie škandinávskej obrany modelom

V počiatočnej sekvencii ťahov `1. e4 d5 2. exd5 Qxd5 3. Nc3 Qa5 4. d4` je možné vidieť, že model ukážkovo reprodukoval šachové otvorenie nazývané Škandinávska obrana (obrázok 6.3). Zápas následne pokračoval štandardným spôsobom, až kým nedošlo k hrubej chybe čierneho hráča ťahom `Nxe5`. Chybu následne zúžitkoval model ťahom `11. Qxd8`, ktorý viedol k matu (obrázok 6.4).



Obr. 6.4: Model vyhráva matom nad súperom

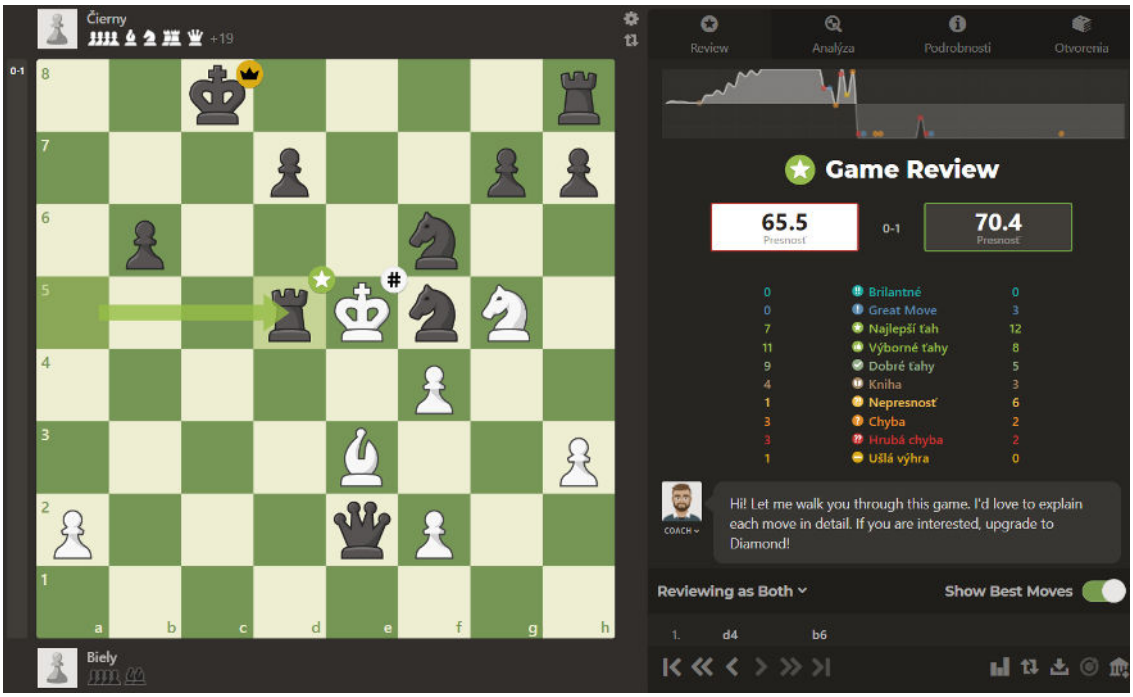
Nasledujúci zápas sa odohrával medzi modelom a Minimax algoritmom hĺbky 4:

```
[Site "Chess.com"]
[Date "2022.04.10"]
[White "Model"]
[Black "Minimax algoritmus"]
[Result "0-1"]

1. d4 b6 2. c4 Bb7 3. Nf3 e6 4. Nc3 Bxf3 5. gxf3 a5 6. e4 Qe7 7. Be3
Kd8 8. Qd2 Nc6 9. O-O-O Ra7 10. Kb1 Kc8 11. d5 exd5 12. cxd5 Nd8 13.
f4 Ra8 14. Bh3 Qh4 15. e5 Qxh3 16. Rhg1 f5 17. Rg5 a4 18. Rxf5 Qxf5+
19. Kc1 Qh5 20. Ne4 Qf3 21. Ng5 Qg4 22. d6 a3 23. bxa3 Qh5 24. Qd4 Nc6
25. Qd2 cxd6 26. exd6 Nh6 27. Ne4 Rxa3 28. Kb1 Qf3 29. Ng5 Qg4 30. h3
Qf5+ 31. Kb2 Ra6 32. Qc3 Qb5+ 33. Qb3 Ra5 34. Ne6 Bxd6 35. Ng5 Qe2+ 36.
Kc3 Bb4+ 37. Qxb4 Nxb4 38. Rd2 Nd5+ 39. Kd4 Nf5+ 40. Ke5 Nf6+ 41. Rd5
Rxd5# 0-1
```

Obr. 6.5: Napodobenie anglickej hry modelom

V tomto zápase model prehral, hrajúc s presnosťou 65,5%. Zo začiatku je znova možné vidieť správne otvorenie modelom v sekvencii `1. d4 b6 2. c4 Bb7 3. Nf3 e6 4. Nc3`, keďže reprodukuje otvorenie nazývané anglická hra (obrázok 6.5). Minimax algoritmus bez znalosti otvorení iba spočiatku vykonáva náhodné ťahy, kým nemá možnosť získavať súperov materiál.



Obr. 6.6: Model prehráva proti Minimax algoritmu

Približne do prvej polovice hry biely vykonáva vynikajúce pozičné ťahy, čo potvrdzuje aj evaluačná krivka. Neskôr však dochádza ku súbojom v jednotlivých častiach šachovnice, ktoré vyhráva Minimax algoritmus (ťah `Qxh3`). Postupným získavaním materiálu Minimax algoritmus otáča hru, čo vedie až k prehre modelu (obrázok 6.6).

Z uvedeného vyplýva, že sa model bol schopný naučiť správanie, ktoré odpozoroval v existujúcich hrách a reprodukoval väčšinu ťahov. Exceloval skôr v počiatočnej fáze hry, a v kratších hrách. Naopak, pri dlhších hrách, ktoré prinášali veľkú variabilitu pozícií už model chyboval a strácal materiál. Pre lepšiu presnosť vykonaných ťahov je teda potrebné využívať aj jeden z vyhľadávacích algoritmov.

7 Obohatenie modelu o vyhľadávanie Minimax

Spôsobom, ako vylepšiť presnosť riešenia vo výbere ťahov je pridanie vyhľadávacieho algoritmu. Riešením tak bude pridanie Minimax algoritmu, ktorý bol opísaný v časti 2.3. Dôvodom výberu tohoto algoritmu je jednoduchosť jeho implementácie, a možná budúca úprava na algoritmus Alfa-Beta, ktorý by bol schopný optimálnejšie vyhľadávať v hernom strome. Keďže algoritmus je založený na vyhľadávaní do hĺbky, kde po dosiahnutí očakávanej hĺbky ohodnotí daný stav (uzol), bolo potrebné implementovať heuristickú funkciu, ktorá bude schopná daný uzol ohodnotiť.

7.1 Heuristická funkcia

```
def get_position_rating(chess_squares, square_to, color):
    piece = chess_squares.piece_at(square_to)
    rating = 0
    if piece is None:
        return 0
    if piece.piece_type == chess.QUEEN:
        rating = 9
    elif piece.piece_type == chess.ROOK:
        rating = 5
    elif piece.piece_type == chess.BISHOP:
        rating = 3
    elif piece.piece_type == chess.KNIGHT:
        rating = 3
    elif piece.piece_type == chess.PAWN:
        rating = 1
    return rating if color == chess.WHITE else -rating
```

Zdrojový kód 7.1: Heuristická funkcia

Heuristická funkcia (zdrojový kód 7.1) bola založená na zistení materiálového rozdielu medzi bielym a čiernym hráčom. Ako už bolo spomínané v teoretickej časti, každý typ figúry má inú silu, a práve tento princíp bol uplatnený v heuristickej funkcii. Výpočet materiálu v každom koncovom uzle by mohol byť časovo náročný, preto je hodnota zisťovaná kumulatívne, stále pri ťahu ktorý vyústi k zmene materiálu. Táto hodnota je stále propagovaná ďalšiemu dcérskemu uzlu, ktorý bude navštívený. Neskôr bola do funkcie pridaná aj podmienka zisťujúca, či pri ťahu dochádza ku premene pešiaka na inú figúru (povýšenie). V tomto prípade by sa hodnota pozície zvýšila o hodnotu novej figúry.

7.2 Vyhľadávanie

Na začiatku vyhľadávania je potrebné získať z aktuálneho herného stavu legálne ťahy funkciou `legal_moves = list(board.legal_moves)`. Keďže vyhľadávanie môže byť časovo náročné, bolo potrebné zvoliť vhodnú hĺbku vyhľadávania. Z praktických dôvodov bola maximálna hĺbka pre vyhľadávania určená na hodnotu 4, čo je kompromisom medzi kvalitou výsledku a časom spracovania. Pre rýchlejší prístup k výsledkom bolo vyhľadávanie paralelizované, čo umožní využitie všetkých vláken procesora. Funkciou `multiprocessing.cpu_count()` bolo možné zistiť počet vláken na danom stroji.

Následne prebieha iterovanie cez všetky získané ťahy, a príprava parametrov pre funkciu vnorenia. Pre každý ťah je spustený jeden samostatný proces. V prípade, že je ťahov viac ako vláken procesora, spustia sa iba procesy, pre ktoré je dostupné voľné vlákno. Po skončení niektorého z procesov sa spustí na danom vlákne nový proces. Tento postup prebieha, kým nedôjde k výpočtu všetkých ťahov. Po skončení vnorených funkcií sa z výsledkov vyberie maximálny, ak je na ťahu biely hráč, alebo minimálny, ak je na ťahu čierny hráč. V prípade, že existuje viacero ťahov s rovnakým maximálnym, resp. minimálnym výsledkom, je pomocou neurónovej siete vybraný taký ťah, ktorý neurónová sieť v predikcii určí ako najpravdepodobnejší.

Spomínaná funkcia vnorenia plní principiálne veľmi podobnú funkciu, aká bola spomínaná vyššie. Táto funkcia už však nerozdeľuje ťahy medzi jednotlivé vlákna, ale pracuje s výpočtami iba na vlákne v ktorom bola vytvorená, a ďalšie vnorenia sú vykonávané tiež v rámci toho istého vlákna.

Ak stav v momentálnej funkcii má dcérske stavy, tie budú rekurzívne vypočítane. Stavy, v ktorých ťah vyberá biely hráč vrátia ako výsledok maximálnu hodnotu svojich dcérskych stavov. Inak je vrátený minimálny výsledok dcérskych sta-

vov.

V prípade, že funkcia narazí na stav, ktorý je mat, vráti hodnotu +1000 ak vyhral biely, resp. -1000 ak vyhral čierny. Táto hodnota bola zvolená z dôvodu, že ohodnotenie uzla v prípade výhry alebo prehry by mala prevyšovať akúkoľvek inú hodnotu, ktorá bola daná materiálom. V prípade remízy je vrátená hodnota 0. Ak nastáva takýto koncový stav, žiadne vnorenie už nepokračuje, a funkcia vráti vypočítanú hodnotu.

V prípade, že ide o stav v maximálnej hĺbke vyhľadávania, je vrátený rozdiel materiálu.

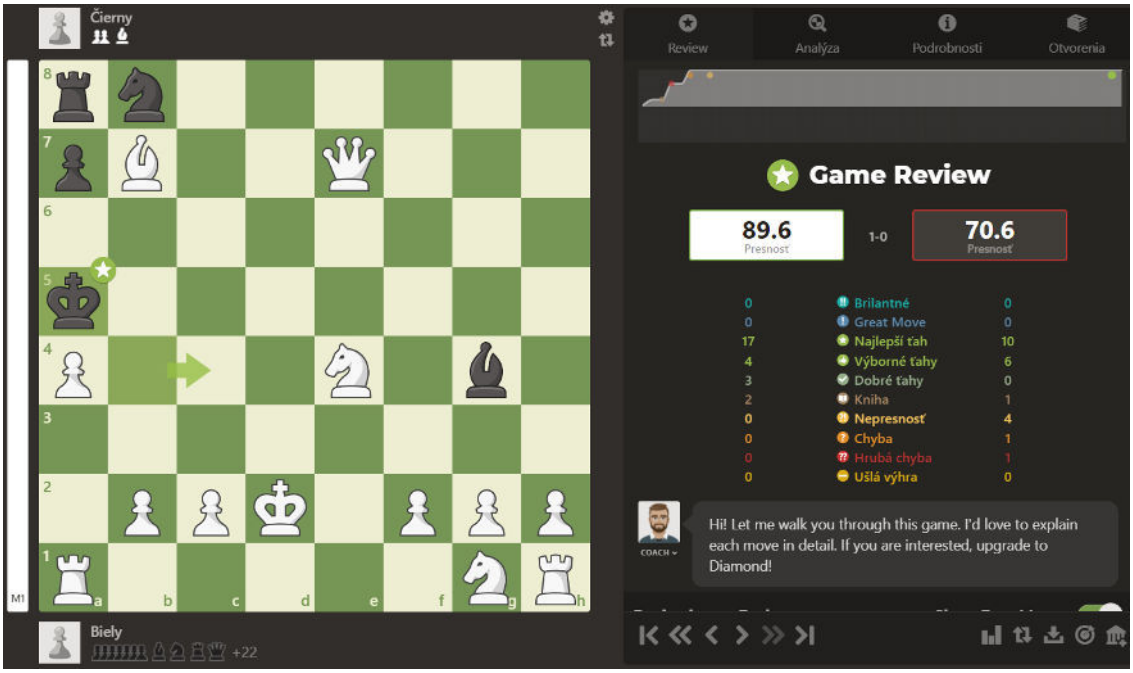
7.3 Overenie riešenia

Overenie riešenia prebiehalo experimentálnym spôsobom, analogickým k sekcii 6.3. V riešení je poznať lepší zmysel pre získavanie a obranu materiálu, čo sa ukázalo aj na lepšej priemernej presnosti ťahov. Kým v čistom modeli bola dosahovaná priemerne 64 percentná presnosť, po pridaní Minimax algoritmu vzrástla priemerná presnosť na necelých 71%. Oproti pôvodnému riešeniu je model podstatne úspešnejší v zápasoch proti Minimax algoritmu bez neurónovej siete. Príkladom je jedna z najpresnejších hier, kedy riešenie porazilo spomínaný algoritmus s presnosťou ťahov 89,6%:

```
[Site "Chess.com"]
[Date "2022.04.10"]
[White "Model+Minimax"]
[Black "Minimax algoritmus"]
[Result "1-0"]

1. e4 f6 2. d4 f5 3. exf5 e5 4. dxe5 c5 5. Qh5+ g6 6. fxg6 Qa5+ 7. Bd2
Nf6 8. exf6 Qxd2+ 9. Kxd2 Kd8 10. g7 Bxg7 11. fxg7 Rg8 12. Qxh7 Rxg7
13. Qxg7 c4 14. Nc3 d6 15. Qf6+ Kc7 16. Bxc4 Bd7 17. Ne4 d5 18. Bxd5
Kc8 19. Nd6+ Kc7 20. Bxb7 Bg4 21. Qe7+ Kb6 22. Qd8+ Kc5 23. Ne4+ Kb5
24. a4+ Kb4 25. Qe7+ Ka5 26. Qc5# 1-0
```

Na obrázku 7.1 je zobrazená pozícia z konca hry, kedy pridaním ťahu `26. Qc5#` toto riešenie modelu skombinované s Minimax algoritmom zvíťazilo.



Obr. 7.1: Výhra nad algoritmom Minimax na jeden ťah

8 Vyhodnotenie

K čiastkovým vyhodnoteniam dosiahnutých výsledkov už došlo v sekciách 5.4, 6.3 a 7.3, kde boli analyzované výsledky daných modelov, a vykonaná prípadová štúdia. V tejto kapitole budú riešenia porovnané, a dôjde k celkovému zhrnutiu výsledkov.

Prvým riešením bol algoritmus na princípoch AlphaZero. To bolo založené na učení s posilňovaním, kde iteratívne dochádza k hrám modelu samého so sebou. Z týchto hier sú za pomoci MCTS simulácií tvorené vzorky, ktoré sú následne poskytnuté konvolučnej neurónovej sieti. Po každom odohraní základných hier a naučení sa na vzorkách nasleduje zápas medzi pôvodným modelom, a modelom trénovaným pomocou nových vzoriek. Hlavným predpokladom pre vyhodnotenie presnosti predikcie modelu je, že proces trénovania prebehne v požadovanej dĺžke, a opakovane nastanú prípady, kedy natrénovaný model prekoná ten predchádzajúci. Kvôli časovej náročnosti výpočtu, kedy tvorba vzoriek z jednej hry trvala viac ako 10 minút, nebolo možné tento predpoklad splniť. Napriek opakovaným pokusom o zníženie výpočtovej náročnosti algoritmu, ktoré sú popísané v danej sekcii 5.4, sa nepodarilo v primeranom čase model naučiť na dosiahnutie požadovaných výsledkov. Ako potenciálne zlepšenie výsledkov je možné v budúcnosti zmeniť algoritmus MCTS zo sekvenčného na paralelný. Inou myšlienkou na vylepšenie by mohlo byť použitie profesionálneho hardvéru, ako tomu bolo pri pôvodnej implementácii AlphaZero.

Nadväzujúcim riešením bola implementácia strojového učenia na princípe imitácie. To pozostávalo zo získania vhodných dát, a ich spracovania v kapitole 4. Neurónová sieť pozostávala zo štyroch konvolučných vrstiev. Učenie prebiehalo samostatne pre biely a čierny model na necelých dvoch miliónoch hrách. Priemer učenia bol meraný chybovou funkciou. Chybovosť pridávaním iterácií klesala, čo je zobrazené v grafe na obrázku 6.1, resp. 6.2. Experimentálnym hraním zápasov a následnou analýzou zápasov bola nameraná priemerná presnosť vykonaných ťahov 64%. Záverom z kapitoly boli poznatky, že imitačné učenie je pomerne presné v predikcii ťahov v počiatočných fázach hry alebo budovaní pozície. Uká-

zala sa však nevýhoda absencie dopredného vyhľadávania, kvôli čomu model strácal materiál a prehrával zápasy.

Keďže algoritmus založený na imitačnom učení bolo potrebné vylepšiť o vyhľadávací algoritmus, bolo pridané vyhľadávanie Minimax (kapitola 7), ktoré je vhodné pre hry dvoch hráčov. Tento algoritmus bol popísaný už v analytickej časti. Následkom bola implementácia heuristickej funkcie, ktorá je schopná v koncových uzloch ohodnotiť herné stavy. Pri každom vnorení rekurzívnej funkcie dochádzalo k výberu maximálnej, resp. minimálnej hodnoty z dcérskych uzlov. Z dôvodu časovej náročnosti bola zvolená hĺbka vyhľadávania 4. Taktiež bola pridaná do riešenia paralelizácia, ktorá urýchli výpočet využitím všetkých vláken procesora. Experimentálnym hraním zápasov a následnou analýzou zápasov bola nameraná priemerná presnosť vykonaných ťahov takmer 71%, čo je znateľné zlepšenie v porovnaní s predchádzajúcim riešením bez vyhľadávania. Algoritmus bol efektívnejší s udržaním svojho materiálu, a získavaním súperovho materiálu.

Pre porovnanie riešení a zistenie, ktoré z implementácií dosahuje najlepšie výsledky bol do riešenia pridaný súbor `Match.py`, ktorý obsahuje implementáciu pre zápasenie medzi algoritmami. Pomocou tohoto python súboru tak je možné určiť počet hier, ktoré majú dva algoritmy medzi sebou odohrať. Pre tento účel bolo experimentálne odohraných 100 hier medzi algoritmami systémom každý s každým. Výsledky sú zobrazené v tabuľke 8.1. Každý z výsledkov je zobrazený vo formáte (výhry : remízy : prehry). Každý z výsledkov je zobrazený z pohľadu algoritmu, ktorý je vymenovaný naľavo v danom riadku.

	Model+Minimax	**Minimax**	**Model**	**Náhodný**
Model+Minimax	**X**	64 : 31 : 5	94 : 6 : 0	95 : 5 : 0
Minimax	5 : 31 : 64	**X**	62 : 28 : 10	91 : 9 : 0
Model	0 : 6 : 94	10 : 28 : 62	**X**	66 : 33 : 1
Náhodný	0 : 5 : 95	0 : 9 : 91	1 : 33 : 66	**X**

Tabuľka 8.1: Výsledky zápasov

Najlepšie výsledky dosiahol podľa očakávaní algoritmus zložený z modelu

(neurónovej siete) a algoritmu Minimax. Dokázal presvedčivo vyhrávať stretnutia proti všetkým typom algoritmov, ako je možné vidieť aj vo vzájomnej bilancii s inými algoritmami. Druhým najúspešnejším bol čistý algoritmus Minimax, ktorý slúžil hlavne referenčný bod na zlepšovanie implementácií v rámci tejto práce. Tretím najlepším bol model bez vyhľadávacieho algoritmu, ktorý dokázal mať pozitívnu bilanciu iba proti náhodnému algoritmu. Pre zaujímavosť, posledný skončil náhodný algoritmus. Ten generuje platné ťahy náhodne, bez akejkoľvek znalosti šachových princípov. Algoritmus bol implementovaný iba pre potreby porovnania v rámci tejto kapitoly. Ako vyplýva z tabuľky, bol porazený všetkými ostatnými implementáciami.

9 Záver

V úvode práce bol čitateľ uvedený do problematiky stručným opisom kľúčových historických míľnikov súvisiacich s umelou inteligenciou v šachu. Následne bola vysvetlená aktuálnosť problému, keďže umelé inteligencie medzi sebou neustále súperia, a sú to práve algoritmy založené na strojovom učení, ktoré majú v týchto zápasoch navrch.

Na začiatku analytickej časti boli vysvetlené základné šachové pojmy a princípy. Kapitola slúži na základné pochopenie hry šach aj pre laika, ktorý s touto hrou doteraz nemal skúsenosti. Bolo tu vysvetlené základné rozloženie šachovnice, sila jednotlivých figúr, aj rôzne druhy zápisov šachových pozícií a ťahov.

Následne bola venovaná pozornosť umelej inteligencii v hrách, keďže hry sú vhodným prostriedkom na štúdium umelej inteligencie, čo môže následne pomáhať v ďalších odvetviach ako vedecký výskum alebo medicína. Ďalej bola pozornosť venovaná vysvetleniu pojmu faktor vetvenia, a ako veľmi ovplyvňuje vyhľadávanie v hrách Go alebo šach. Pomocou grafu bolo vysvetlené, akým spôsobom sa vypočítal faktor vetvenia v hre šach. Následne boli vysvetlené rôzne druhy vyhľadávacích algoritmov, a optimalizácie využívané vyhľadávaní stavov v šachu.

V analytickej časti došlo tiež k porovnaniu existujúcich šachových simulátorov a vysvetleniu ich princípov. Boli rozoberané riešenia Stockfish a AlphaZero, ktoré patria medzi kľúčové riešenia v tejto problematike. Skúmal sa spôsob, akým jednotlivé algoritmy vykonávajú vyhľadávanie a evaluáciu šachových pozícií. Nakoniec boli riešenia porovnané z pozície ELO hodnotenia, v ktorých mali mierne lepšie výsledky práve algoritmy založené na AlphaZero.

Praktická časť začala implementáciou používateľského rozhrania pre hru šach. Informácie na prácu s používateľským rozhraním budú zahrnuté v používateľskej príručke. Nasledovala kapitola zameraná na predspracovanie dát, v ktorej boli získané takmer dva milióny šachových partií. Dáta boli neskôr spracované pomocou funkcií popísaných v rámci kapitoly.

Ako prvé riešenie pre tvorbu umelej inteligencie v šachu bola implementácia založená na učení s posilňovaním AlphaZero. Riešenie pozostávalo z algoritmu

hrajúceho hry samého sebou, a následného trénovania konvolučnej neurónovej siete. Tento proces sa dial iteratívne, kde na konci každej iterácie došlo k zápasu medzi pôvodným a trénovaným modelom. Tento prístup sa ukázal ako pomerne časovo náročný, a nepriniesol znateľné výsledky. Budúcim riešením by mohlo byť pridanie paralelných procesov, alebo využitie výkonnejšieho hardvéru.

Nadväzujúcim riešením bola implementácia strojového učenia na princípe imitácie, kde boli využité dopredu spracované dáta. Tento algoritmus už bol schopný vykonať trénovanie v dostupnom čase, a priniesol postačujúce výsledky. Natrénovaný model bol pomerne presný v určovaní ťahov v otvorení a krátkych hrách, kedy bol familiárny s pozíciou. V neskorších fázach hry už dochádzalo k chybám a strate materiálu.

Vylepšením implementácie bolo pridanie vyhľadávania Minimax do riešenia, čo vyústilo k lepšej priemernej presnosti vybraných ťahov. V kapitole zameranej na vyhodnotenie riešenia prebiehali zápasy medzi viacerými algoritmami, kde táto posledná implementácia v bilanciách výrazne prekonala iné druhy algoritmov. Kým pôvodný model mal pri zápasoch s algoritmom Minimax negatívnu bilanciu, model obohatený o vyhľadávanie mal proti algoritmu Minimax už jednoznačne pozitívnu bilanciu.

Pri implementácii došlo k merateľnému pokroku v precíznosti ťahov, ktoré algoritmus produkoval. Imitačné učenie tak určite má svoje zastúpenie pri implementácií a trénovaní umelej inteligencie. Predikcia z neurónovej siete však nemusí vždy stačiť, a je vhodné ju dopĺňať vhodným vyhľadávacím algoritmom. Pre zlepšenie výsledkov tak, aby sa blížili úrovni profesionálnych simulátorov je potrebné do budúcna vykonať niekoľko krokov. Išlo by napríklad o použitie výkonnejšieho hardvéru, využitie paralelných procesov vo vyhľadávaní, alebo zmena aktuálneho Minimax vyhľadávania na Alfa-beta orezávanie, ktoré by malo byť podľa teórie optimálnejšie. Za zváženie tiež stojí voľba iných vrstiev a hyperparametrov pri trénovaní neurónovej siete.

Literatúra

1. TURING, Alan; BOWDEN, B.V. *Faster Than Thought - A Symposium on Digital Computing*. 1. vyd. Sir Isaac Pitman & Sons, 1953.
2. *IBM100 - Deep Blue* [https://www.ibm.com/ibm/history/ibm100/us/en/icons/deepblue/]. [B.r.]. (Navštívené dňa 03/19/2022).
3. *Chess - Wikipedia* [https://en.wikipedia.org/wiki/Chess]. [B.r.]. (Navštívené dňa 04/16/2022).
4. REEVE, JUSTIN. *Chess Surges In Popularity Since The Pandemic* [https://www.thegamer.com/chess-booms-in-popularity-during-the-pandemic/]. 2021. (Navštívené dňa 04/19/2022).
5. *Šachmat – Wikipédia* [https://sk.wikipedia.org/wiki/%C5%A0achmat]. [B.r.]. (Navštívené dňa 04/16/2022).
6. FIDE. *Pravidlá šachu FIDE* [https://www.chess.sk/download/dokumenty/153381608815148018 37Pravidla_sachu_FIDE_platne_od_Jan-01-2018_cistaverzia.pdf]. [B.r.]. (Navštívené dňa 04/19/2022).
7. FRANKENFIELD, Jake. *Artificial Intelligence (AI) Definition* [https://www.investopedia.com/terms/a/artificial-intelligence-ai.asp]. [B.r.]. (Navštívené dňa 04/10/2022).
8. LUCAS, Simon M. Computational Intelligence and AI in Games: A New IEEE Transactions. *IEEE Transactions on Computational Intelligence and AI in Games*. 2009, roč. 1, č. 1, s. 1–3. Dostupné z DOI: 10.1109/TCIAIG.2009.2021433.
9. LEVINOVITZ, Alan. *The Mystery of Go, the Ancient Game That Computers Still Can't Win | WIRED* [https://www.wired.com/2014/05/the-world-of-computer-go/]. [B.r.]. (Navštívené dňa 04/10/2022).

10. BARNES, David. *statistics - What is the average number of legal moves per turn? - Chess Stack Exchange* [https://chess.stackexchange.com/questions/23135/what-is-the-average-number-of-legal-moves-per-turn]. [B.r.]. (Navštívené dňa 04/10/2022).

11. *Faktor větvení* [https://wikijii.com/wiki/Branching_factor]. [B.r.]. (Navštívené dňa 04/14/2022).

12. *KvK – Syzygy endgame tablebases* [https://syzygy-tables.info/]. [B.r.]. (Navštívené dňa 04/14/2022).

13. GURARI, Eitan. *CIS 680: DATA STRUCTURES: Chapter 19: Backtracking Algorithms* [https://web.archive.org/web/20070317015632/http://www.cse.ohio-state.edu/~gurari/course/cis680/cis680Ch19.html#QQ1-51-128]. 1999. (Navštívené dňa 04/15/2022).

14. NÁVRAT, Pavol; BEŇUŠKOVÁ, Ľubica; BIELIKOVÁ, Mária; KAPUSTÍK, Ivan; KOSKOVÁ, Gabriela; POSPÍCHAL, Jiří. *Umelá inteligencia*. 2. vyd. Vydavateľstvo STU, 2006. ISBN 80-227-2354-1.

15. NOGUEIRA, Nuno. *Minimax - File:Minimax.svg - Wikimedia Commons* [https://commons.wikimedia.org/wiki/File:Minimax.svg#/media/File:Minimax.svg]. [B.r.]. Licencia: CC BY-SA 2.5 (Navštívené dňa 04/15/2022).

16. JEZ9999. *https://upload.wikimedia.org/wikipedia/commons/9/91/AB_pruning.svg* [https://upload.wikimedia.org/wikipedia/commons/9/91/AB_pruning.svg]. [B.r.]. Licencia: CC BY-SA 3.0 (Navštívené dňa 04/15/2022).

17. *Computer Chess Rating Lists*. [B.r.]. Dostupné tiež z: https://computerchess.org.uk/ccrl/4040/index.html. (Navštívené dňa 03/17/2022).

18. CHAMPION, Antoine. *Dissecting stockfish part 1: In-depth look at a chess engine*. Towards Data Science, 2021. Dostupné tiež z: https://towardsdatascience.com/dissecting-stockfish-part-1-in-depth-look-at-a-chess-engine-7fddd1d83579.

19. BROWNE, Cameron. Bitboard Methods for Games. *ICGA journal*. 2014, roč. 37, s. 70. Dostupné z DOI: 10.3233/ICG-2014-37202.

20. MAHARAJ, Shiva; POLSON, Nick; TURK, Alex. Chess AI: Competing Paradigms for Machine Intelligence. 2021, s. 4–6.

21. *AlphaZero - Chess Engines - Chess.com* [https://www.chess.com/terms/alphazero-chess-engine]. [B.r.]. (Navštívené dňa 03/17/2022).

22. SILVER, David; HUBERT, Thomas; SCHRITTWIESER, Julian; ANTONOGLOU, Ioannis; LAI, Matthew; GUEZ, Arthur; LANCTOT, Marc; SIFRE, Laurent; KUMARAN, Dharshan; GRAEPEL, Thore et al. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*. 2018, roč. 362, č. 6419, s. 1140–1144.

23. SCHULZ, André. *FIDE ratings February 2022 | ChessBase* [`https://en.chessbase.com/post/fide-ratings-february-2022`]. [B.r.]. (Navštívené dňa 03/18/2022).

24. PYGAME DEVELOPMENT TEAM. *About - pygame wiki* [`https://www.pygame.org/wiki/about`]. [B.r.]. (Navštívené dňa 04/14/2022).

25. JOHNPABLOK. *Chess Pieces and Board squares | OpenGameArt.org* [`https://opengameart.org/content/chess-pieces-and-board-squares`]. [B.r.]. (Navštívené dňa 04/03/2022).

26. *Lichess Elite Database – Learn from the strongest!* [`https://database.nikonoel.fr/`]. [B.r.]. (Navštívené dňa 04/03/2022).

27. NAIR, Surag. *Simple Alpha Zero* [`http://web.stanford.edu/~surag/posts/alphazero.html`]. 2017. (Navštívené dňa 04/21/2022).

28. NAIR, Surag [`https://github.com/suragnair/alpha-zero-general`]. [B.r.]. (Navštívené dňa 04/21/2022).

29. *Chess Analysis Board and PGN Editor - Chess.com* [`https://www.chess.com/analysis`]. [B.r.]. (Navštívené dňa 04/21/2022).

Zoznam príloh

Príloha A Používateľská príručka

Príloha B CD médium – záverečná práca v elektronickej podobe

Technická univerzita v Košiciach
Fakulta elektrotechniky a informatiky

Príloha A: Používateľská príručka

2022 **Bc. Pavol Kacej**

Obsah

Zoznam obrázkov

1 Úvod

Používateľská príručka bude slúžiť na opísanie používateľského rozhrania (ďalej nazývaného aj ako "program"), ktoré bolo implementované v rámci diplomovej práce *Šachový simulátor využitím pokročilých metód umelej inteligencie*. Rovnako bude opísaný postup práce s používateľským rozhraním. V prvej časti dôjde k opisu jednotlivých grafických elementov bežiaceho programu. V druhej časti bude vysvetlené, ako prebieha spustenie programu predvolene, alebo kombináciou parametrov daných do konzoly. V poslednej časti bude vysvetlené, akým spôsobom je možné s používateľským rozhraním interagovať, či už pomocou myši, alebo klávesov, ktoré sú podporované. Pohľad na používateľské rozhranie je zobrazený na obrázku 1.1.

Obr. 1.1: Základné postavenie šachovnice v používateľskom rozhraní

2 Opis používateľského rozhrania

Používateľské rozhranie po štarte zobrazí grafické okno, na ktorom je začiatočná šachová pozícia pripravená na zápas. Okno je predvolené na rozlíšenie *900x800*. V prípade, že je nastavený ako biely hráč jeden z algoritmov, je potrebné počkať kým vykoná prvý ťah a až tak bude možné interagovať so šachovnicou. Hra začína predvolene z pozície čierneho hráča. Ako pozadie je aplikovaný obrázok drevenej dosky, pre imitáciu fyzickej šachovnice.

Na ľavej strane šachovnice sú vertikálne zobrazené čísla radov šachovnice určené pre ľahké zorientovanie používateľa, alebo pre uľahčenie notácie ťahov. Obdobne sú na spodnej strane rozhrania písmenami abecedy označené stĺpce. Na pravej strane šachovnice je pre oboch hráčov zobrazené číslo popisujúce, či má hráč materiál vo vyššej hodnote ako súper alebo naopak. Ak dôjde pri hre k odstráneniu jednej z figúr, bude táto figúra zobrazená na pravej strane rozhrania, pri konkrétnom hráčovi.

3 Spustenie používateľského rozhrania

Hlavný súbor jazyka `python`, ktorý je zodpovedný za spustenie používateľského okna a hry sa nazýva `Chess.py`. Je umiestnený v priloženom zdrojovom kóde v podpriečinku `chessgame`.

3.1 Prvotné spustenie programu

Pre prvé spustenie programu potrebné otvoriť príkazový riadok daný v konkrétnom operačnom systéme. Následne je potrebné sa v príkazovom riadku navigovať do koreňového priečinka projektu. Konzolovým príkazom:

```
cd chessgame
```

používateľ vojde do cieľového priečinku. Následne je možné príkazom:

```
python .\Chess.py
```

spustiť program. Keďže neboli dané žiadne argumenty programu, program sa spustí s vopred definovanými parametrami hry. Predvolene bude hrať za bieleho model obohatený o algoritmus Minimax, a čierny hráč bude človek.

3.2 Argumenty príkazového riadku

Program však umožňuje používateľovi pomocou argumentov nadefinovať niektoré aspekty hry. Podobne ako tomu bolo v texte diplomovej práce, aj tu program rozlišuje **viacero typov hráčov**:

- `human` - šachový hráč ovládaný človekom,
- `minimax` - algoritmus Minimax vyhľadávajúci do danej hĺbky,

- `model` - predikcia neurónovej siete,
- `combined` - kombinácia neurónovej siete a algoritmu Minimax,
- `random` - algoritmus tvoriaci náhodné ťahy.

V prípade, že používateľ chce nastaviť aké typy hráčov budú súperiť, môže tak urobiť iba pre jedného (bieleho hráča), alebo pre oboch z nich. Prvý argument obsahujúci hráča nastaví tomuto hráčovi bielu farbu. Druhý argument obsahujúci hráča nastaví tomuto hráčovi čiernu farbu. Príklad spustenia programu definovaním jedného hráča môže vyzerať nasledovne:

```
python .\Chess.py minimax
```

Pre zaujímavosť je možné nastaviť aj oboch hráčov na ten istý typ, a sledovať výsledok zápasu. Príkladom by mohlo byť spustenie:

```
python .\Chess.py random random
```

Ako už bolo spomínané v sekcii 3.1, ak hráči pomocou argumentu nie sú uvedení, predvolene je biely hráč typu `combined` a čierny hráč typu `human`.

Programu je možné pomocou argumentov nastaviť tiež **hĺbku vyhľadávania** pomocou celého čísla. Tento argument nie je povinný, a má vplyv iba ak sa medzi hráčmi nachádza hráč typu `minimax` alebo `combined`, ktoré využívajú hĺbkové vyhľadávanie. Predvolene je hĺbka vyhľadávania nastavená na hodnotu 4. Príkladom spustenia zápasu s hĺbkou 3 by bol príkaz:

```
python .\Chess.py combined human 3
```

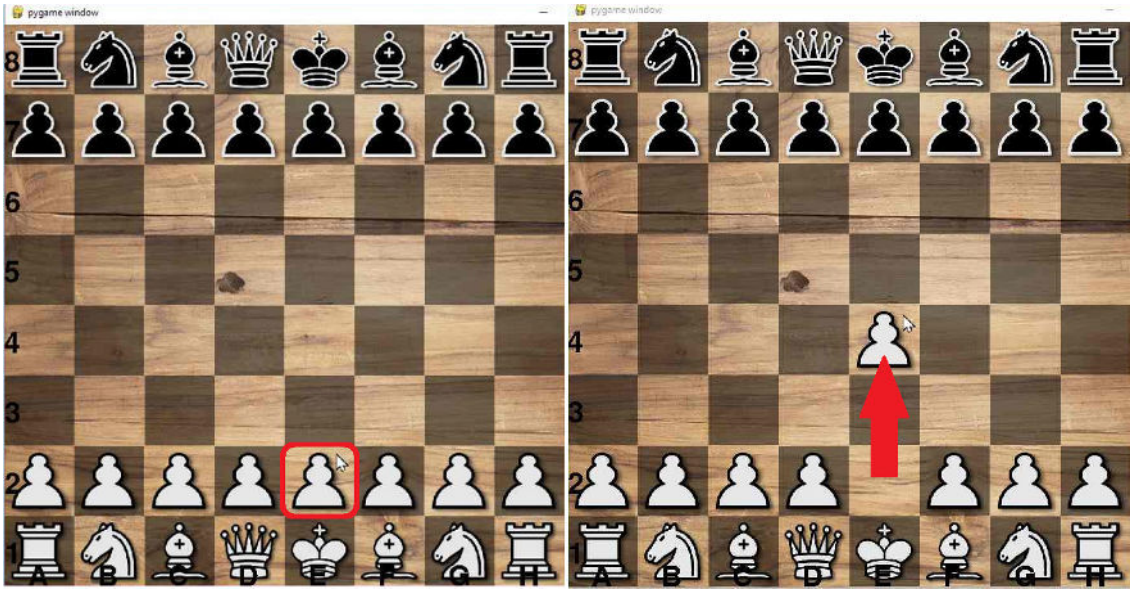
Posledným argumentom je **počet jadier**, ktoré budú použité na vyhľadávanie. Aj v tomto prípade ide o celé číslo, a má vplyv na výkon programu iba ak hra obsahuje jeden z algoritmov využívajúcich vyhľadávanie. V prípade, že hodnota nie je definovaná pri spustení, program zistí počet jadier na danom stroji, a využije všetky z nich. **Využitie všetkých jadier na výpočet môže byť zdrojovo náročné, preto je potrebné v prípade chyby znížiť počet jadier alebo hĺbku vyhľadávania.** Príklad spustenia programu s využitím štyroch jadier by mohol vyzerať nasledovne:

```
python .\Chess.py combined human 3 4
```

4 Ovládanie používateľského rozhrania

Medzi základné interakcie, ktoré je potrebné na šachovnici vykonať, je vykonanie ťahu, respektíve pohyb figúrou. Ten je implementovaný štandardne, spôsobom podobným ťahu rukou na fyzickej šachovnici. Pohyb vlastnou figúrou je možné vykonať kliknutím ľavého tlačidla myši na štvorci, kde sa nachádza figúra, a následným potiahnutím a pustením tlačidla na cieľovom štvorci. V prípade, že ide o valídny ťah, bude tento ťah vykonaný. Tento spôsob je zobrazený na obrázku 4.1.

Druhým spôsobom pohybu figúrou je kliknutie na štartovacej pozícii (aj s následným pustením tlačidla myši), čím sa figúra v programe označí, a následným kliknutím na cieľovú pozíciu.



Obr. 4.1: Zachytenie figúry myšou a pustenie myši na cieľovej pozícii

4.1 Klávesové skratky

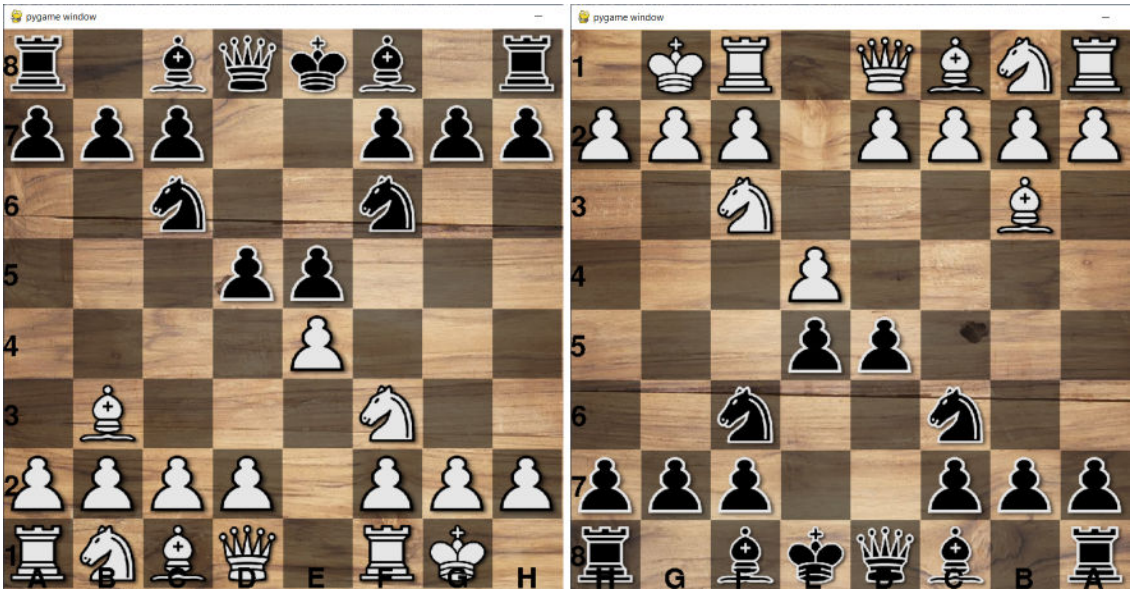
Pri hraní šachovej partie hry môže nastať prípad, kedy je potrebné nazrieť do histórie a zistiť, ako vyzerala staršia pozícia. To je možné pomocou pomocného výpisu v konzole, ktorý program produkuje. Pri každom ťahu je generovaná notácia ťahov, ktorá slúži vo svojej podstate ako export šachovej hry. Iným spôsobom, ako je možné navigovať históriou je pomocou stlačenia klávesy:

- **šípka doľava**, ktorá naviguje na predchádzajúci ťah, a
- **šípka doprava**, ktorá naviguje na nasledujúci ťah.

Stlačením **klávesy S**, ktorá je odvodená od skratky SVG, je možné zobraziť obrázok poslednej pozície ako obrázok vysokej kvality typu SVG. Príklad je zobrazený na obrázku 4.2.

Obr. 4.2: Vytvorenie SVG okna s pozíciu, v pozadí používateľské rozhranie

V prípade, že používateľovi nevyhovuje predvolené zobrazenie, alebo preferuje pri hre dvoch hráčov menenie zobrazenia, stlačením **klávesy R** obráti pohľad na šachovnicu. Výber klávesy je odvodený od anglického "rotate", teda otočiť. Otočenie pohľadu je zobrazené na obrázku 4.3.



Obr. 4.3: Otočenie z pohľadu bieleho na pohľad čierneho hráča