Střední průmyslová škola a Vyšší odborná škola, Písek, Karla Čapka 402, Písek

programator

Maturitní práce

Šachový bot

Téma číslo 23

autor:

Ondřej Polanecký, B4.I

vedoucí maturitní práce:

Mgr. Milan Průdek

Písek 2020/2021

Anotace

Tato maturitní práce představuje můj šachový engine a vysvětluje některé techniky, které šachoví boti využívají. Také obsahuje GUI s návodem na používání, aby si každý mohl snadno vyzkoušet zahrát si proti šachovému botu.
Klíčová slova: Šachy, Šachový bot

Annotation

This graduation work presents my chess engine and explains some techniques used by chess engines. It aslo contains GUI with guide how to use it, so everyone can easily play against my chess bot.
Keywords: Chess, Chess bot, Chess engine

Poděkování

Obsah

Kapitola 1

Úvod

Tato maturitní práce představuje můj šachový engine a a jak jsem ho vytvořil. Budu se snažit popsat různé techniky a alogoritmi, které šachové enginy využívají.

V této práci nebudu popisovat pravidla šachů, budu počítat s vaší znalostí pravidel. Také nebudu popisovat základní datové struktury, nebo algoritmi jako je binární vyhledávání nebo hashmapa.

Kapitola 2

Popis a struktura šachového enginu

2.1 Teorie her

Šachy jsou stálé nevyřešená hra tzn. neví se jak vypadá bezchybná hra. Jsou nevyřešené, protože počet možných pozic je enormní. Už jenom po třech pohybech je 8 902 možných variant hry. Pro představu jsem vygeneroval graf pomocí programu Graphviz[?], který ukazuje jak se hra po třech tazích větví. Viz obr. A.1

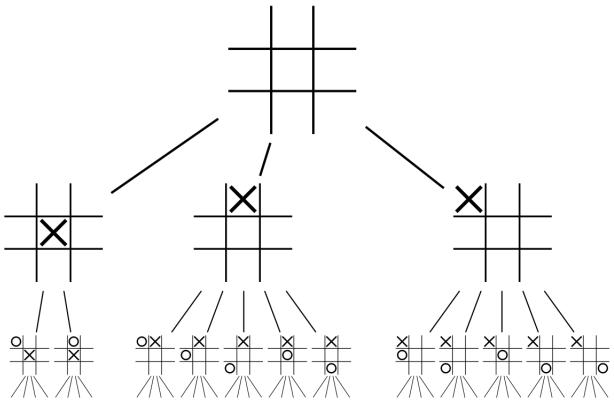
Průměrný počet pohybů ve všech situací je přibližně 35, průměrná hra má 80 pohybů, takže když chceme dostat hodně nepřesný odhad možných šachovým partií, tak nám stačí tyto dvě hodnoty umocnit $35^{80} \approx 10^{123}$[1]. Z tohoto enormního čísla vyplývá, že šachy nelze vyřešit hrubou silou a pravděpodobně šachy v nejbližší době, či dokonce nikdy nevyřešíme. Pro zajímavost dáma má přibližně $5 * 10^{20}$ variací her a byla vyřešena v roce 2007.[13] Při perfektním zahrání od obou hráčů skončí hra remízou.
Šachy jsou:

- hra s úplnými informacemi, tzn. oba hráči ví o všech informacích ve hře (vidí všechny figurky). Na druhou stranu u her s neúplnými informacemi (např. poker), všechny informace neznáme a engine by musel počítat s pravděpodobnostmi pro určité informace a na základě těchto pravděpodobností se rozhodovat.
- hra s nulovým součtem, tzn. jakoukoliv výhodu hráč získá na úkor protihráče.

Díky tomu můžeme tvořit herní strom všech možných tahů do určité hloubky a relativně dobře odhadovat hodnoty pohybů. Herní strom vypadá přibližně takto. Viz obr. 2.1

2.2 Dělení šachových enginů

Zjednodušeně se dají šachové enginy rozdělit do dvou skupin.



Obrázek 2.1: herní strom piškvorek [10]

- Šachový bot založený na alfa-beta vyhledávání
  - Šachový engine prochází rekurzivně herní strom do určité hloubky a na konečných uzlech stromu zhodnotí pozici.
  - Hodnocení pozic je děláno funkcí vytvořenou programátorem a nedostatky evaluační funkce jsou doháňeny prohledáváním spousty pozic do velké hloubky.
  - Detaily budou v kapitole 3.3
- Šachový bot založený na neuronových sítí a prohledávání stromu metodou Monte Carlo
  - Šachový engine prochází rekurzivně herní strom do určité hloubky a na konečných uzlech stromu zhodnotí pozici.
  - Hodnocení pozic je děláno neuronovou sítí, která byla natrénována na hraním proti sobě.
  - Na vyhledávání se používá Monte-Carlo tree search. Oproti Alfa-Beta vyhledávání prohledává podstatně méně pozic, ale prohledává pouze pozice s velkou šancí na úspěšnost.

2.3 Struktura šachového enginu

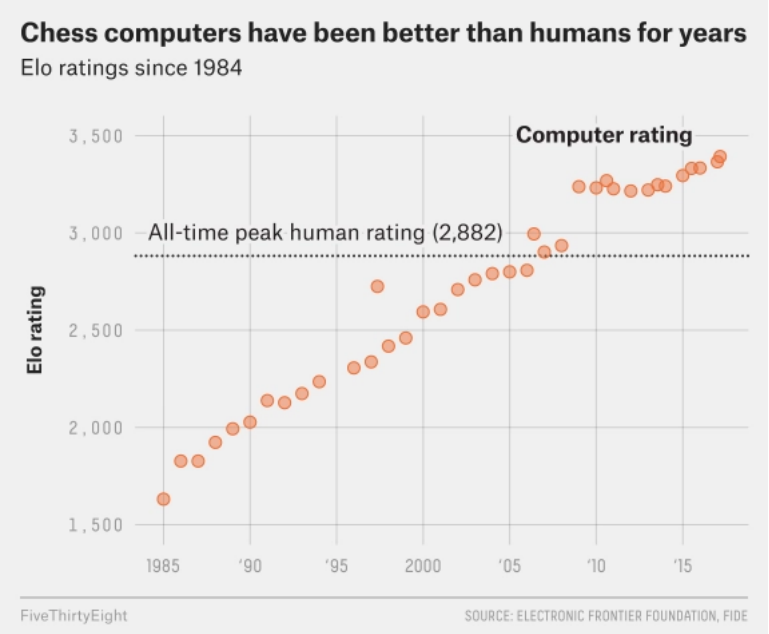
Můj šachový engine bude založen na alfa-beta vyhledávání, takže budu popisovat hlavně tento typ enginů.

Šachový engin by měl mít:

- Generátor legálních pohybů
  - Třída má na starosti generování legálních pohybů v dané situaci
  - Většina šachových enginů nejdřív vygeneruje všechny pseudo-legální pohyby[1], otestuje pohyby a vyřadí u kterých je vlastní král v šachu.
- Třída na reprezentaci stavu hry
  - Udržuje informace o pozicích, právech figurek a případně s nimi hýbe
  - na udržování pozic se většinou používají bitboardy[2]. Detaily o bitboardech v kapitole 3.1
- Transpoziční tabulka
  - Hašovací tabulka která obsahuje hashe pozic a nejlepší pohyb pro určitou pozici.
  - Může obsahovat již prohledané pozice nebo pozice získané z předem vytvořené databáze pohybů.
  - Detaily v kapitole **??**
- Evaluace pozic
  - Hodnotí pozici podle hodnoty, pozice a struktury figurek.
  - Klíčová pro chování šachového enginu, protože na základě ní se rozhoduje v prohledávání pozic.
  - Detaily v kapitole **??**
- Třída na prohledávání pohybů
  - Prohledává strom hry a vrací nejlepší nalezený pohyb.
  -

[1] Pohyby, které odpovídají tomu, jak se mají figurky hýbat, ale je u nich možnost, že by dostali svého krále do šachu.

[2] Bitboard nebo bitmapa je 64 bitové číslo kde v jeho binární podobně každá jednička znamená zabranou pozici na šachovém poli.



Obrázek 2.2: Vývoj Ela šachových enginů[11]

2.4 Historie Šachových enginů

První plně funkční šachové enginy se začaly objevovat v 60. letech minulého století. Algoritmy pro šachové enginy už existovali dřív, ale jejich použití bylo brzděno tehdejším hardwarem. Už v 50. letech minulého století se znali všechny algoritmy, aby mohl být stvořen použitelný šachový engine. Sílá šachových enginů poté prudce rostla zaprvé kvůli Mooreově zákonu[3] a zadruhé díky zlepšování šachových algoritmů. V obr. 2.2 vidíme vývoj Elo[4] hodnocení šachových enginů. Mezi největší inovace za poslední léta patří vydání AlphaZero, které vneslo do dnešních šachových enginů neuronové sítě. Už i tradiční šachový engine jako je Stockfish začal používat na hodnocení pozic neronovými sítě a při vydání této změny získal více než 80 Elo bodů.[15]

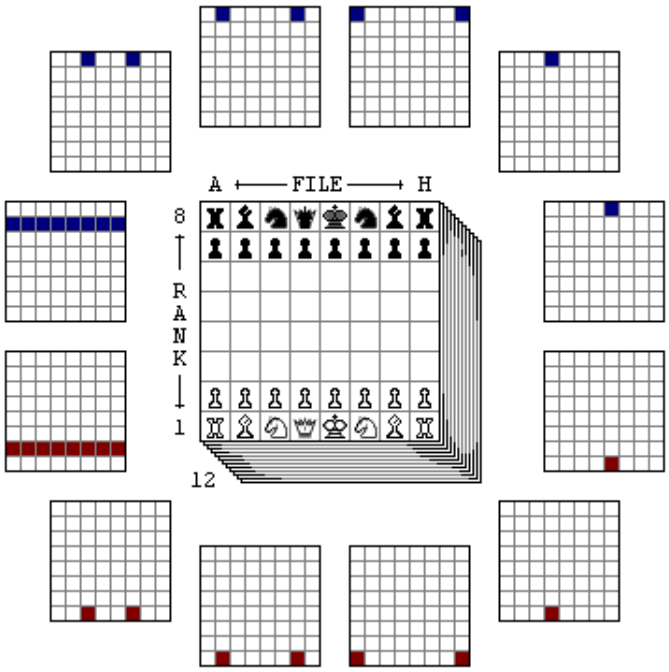
[3]Vypozorovaný zákon, že každé dva roky se zdvojnásobí hustota tranzistorů v integrovaném obvodu.
[4]Statistické ohodnocení výkonnosti hráče

Kapitola 3

Implementace

3.1 Bitboardy

Bitboard je 64 bitové číslo. Každá pozice bitu koresponduje s pozicí na herní ploše. Pokud je pozice na herní ploše zabrána, tak je korespondující bit v bitboardu nastaven na 1. Aby se dali rozlišit jednotlivé figurky, je třeba udržovat v paměti bitboard pro každý druh a barvu figurek. Viz ilustrační obrázek 3.1 Důvod proč se využívají bitboardy je kvůli rychlosti generování pohybů. Například u generování pohybů pro koně jsme schopni si pro každou pozici předpočítat možné pohyby koně. Také můžeme na bitboardy používat logické operace, takže můžeme například všechny pěšáky posunout o 8 bitů doprava (pohyb nahoru) v jednom clock cyclu. V c++ jsem použil datovou strukturu unsigned long long.



Obrázek 3.1: Reprezentace herní plochy pomocí bitboardů[6]

V kódu udržuji bitboardy ve třídě Board. Navíc k tomu mám metody, které určité bitboardy spojí logickou operací OR. Takhle vypadá definice třídy.

```
typedef unsigned long long bitboard;
class Board {
    public:
    bitboard all_bitboards[2][6]{};

    // spojene pozice

    // vraci bitboard figurek urcite barvy
    bitboard PiecesOfColor(bool color);

    // vraci bitboard vsech figurek
    bitboard AllPieces();
}
```

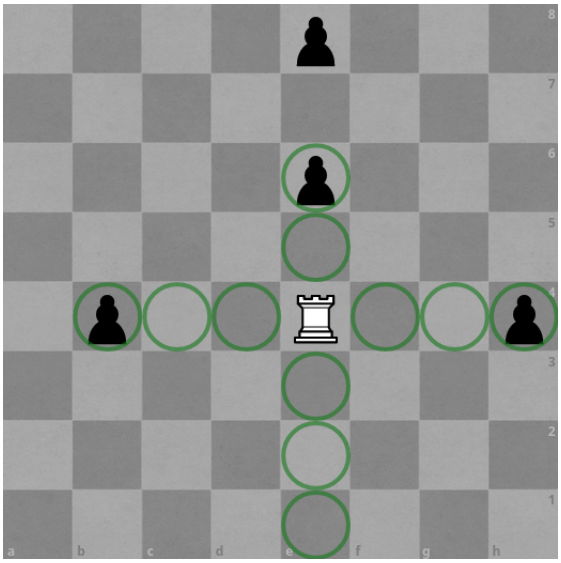
3.2 Generování pohybů

Generování pohybů při používání bitboardů se dělá specifickým způsobem. Pro figurky, které se hýbou nezávisle na tom kde jsou postaveny ostatní figurky, je to lehké. Pouze

pro všech 64 pozic kde se může figurka nacházet si předpočítáme možné pozice kam může jít. Všechny tyto pozice uložíme do pole a jako index použijeme pozici bitu. Když potom narazíme na figurku na určitém poli, stačí vzít předpočítaný bitboard. Generování potom probíhá postupným procházení předpočítaného bitboardu. Aby bylo procházení bitboardu co nejrychlejší používám funkci __builtin_ffsll(). Tato funkce je závislá na kompilátoru (GCC) a snaží se použít přímo assembly instrukci, pokud je dostupná. Funkce vrátí velmi rychle index nejméně významného bitu, který je jedna. Tím dostanu index pozice kam může figurka jít. Prohledaný bit poté nastavím na nulu abych příště dostal další jedničku.

Na ukázku přikládám úryvek kódu pro generování pohybů koňem.

```
1     bitboard enemy_pieces = board.EnemyPieces();
2     bitboard my_pieces = board.MyPieces();
3     bitboard my_knights = board.all_bitboards[board.on_turn][KNIGHT];
4     int from;
5     int to;
6     while (my_knights) {
7         // nacteni pozice dalsiho kone
8         from = __builtin_ffsll(my_knights);
9         from--;
10        // smazani nacteneho kone
11        my_knights ^= 1ULL << (from);
12        // utoky ze soucasneho kone
13        bitboard attacks = precomp.precomputed_knights[from];
14        // odstraneni pohybu, ktere by skoncili na nejake me figurce
15        attacks &= ~my_pieces;
16        while (attacks) {
17            //nacteni pozice pro pohyb
18            to = __builtin_ffsll(attacks);
19            to--;
20
21            // smazani nactene pozice
22            attacks ^= 1ULL << (to);
23            if (enemy_pieces >> to & 1ULL) {
24                // pohyb sebere figurku
25                moves.push_back(Move{from, to, KNIGHT, board.getPieceAt(to), true});
26            }
27            else {
28                // pohyb nesebere figurku
29                moves.push_back(Move{from, to, KNIGHT});
30            }
31        }
```



Obrázek 3.2: Možnosti pohybu věže

Vetší problém je si předpočítat pohyby pro figurky u kterých záleží na jakých pozicích jsou ostatní figurky (věž, střelec, dáma). Je snadné Když si vezmeme třeba tento příklad.3.2

Vidíme, že nás tedy zajímá jaké figurky blokují pohyb věže. Můžeme si teda spočítat všechny možnosti kde můžou být. Zajímají nás jenom figurky, které jsou na V tomto konkrétním případě je to 14 pozic kde můžou být figurky umístěny. To je $2^{14} = 16384$ pozic jak můžou být figurky uspořádány. Dokonce to číslo můžeme ještě zmenšit, protože na úplně krajních pozicích nezáleží. Krajní pozice totiž nic neblokují. Takže v tomto případě to je $2^{10} = 1\,024$ pozic. Což v dnešní době není problém v paměti udržet. Teď ale musíme vyřešit jak budeme pozice v poli indexovat. Jedna možnost je použít hashmapu a jako index použít bitboard blokujících figurek. Je tu ale rychlejší a paměťově méně náročnější možnost. Jsou to tzv. magické bitboardy. Jde o to, že v celém bitboardu nás zajímají pouze určité bity. A my je můžeme rychlou metodou vyjmout z bitboardu a indexovat pouze podle těchto určitých pozic. Dělá se to tak, že se určitým číslem (magickým) vynásobí bitboard a to číslo je tak šikovné, že nám posune chtěné pozice na prvnich x bitů. V minulém případě by to posunulo těch 10 bitů na začátek čísla. Pak jenom ořízneme zbytek bitboardu a indexujeme podle těchto 10 bitů. Tyto čísla se získávají náhodným zkoušením a hledá se takové aby to namapovalo námi chtěné bity na co nejméně bitů. V ideálním případě na tolik bitů kolik bitů máme. Já jsem si svoje čísla sám nepočítal.

Pouze jsem vzal čísla, které už někdo spočítal[12]

Ukážu na příkladu jak by se vytvořil index pro tuto situaci.

```
bitboard bitů,                    bitboard obsazených                    bitboard
které nás zajímáají                      pozic                  blokujících figurek
. . . . . . . .                   1 1 . . . 1 . .                 . . . . . . . .
. . . 1 . . . .                   . 1 1 1 1 . . 1                 . . . 1 . . . .
. . . 1 . . . .                   . 1 . 1 . . . 1                 . . . 1 . . . .
. . . 1 . . . .                   . . . . . . . .                 . . . . . . . .
. 1 1 P 1 1 1 .         &         . 1 . . . . 1 .         =       . 1 . . . . 1 .
. . . 1 . . . .                   . . . 1 . . . .                 . . . 1 . . . .
. . . 1 . . . .                   . . . . . 1 . .                 . . . . . . . .
. . . . . . . .                   1 . . . . . . .                 . . . . . . . .
```

Vezmeme bitboard bitů, které nás zajímají a provedeme bitový součin s bitboardem obsazených pozic. Tímto dostaneme bitboard blokujících figurek, který už máme vyřešený v našem předpočítaném poli. Jenom z něho musíme dostat co je to za index.

```
 . . . . . . . .                                  1 1 . 1 . . 1 1
 . . . 1 . . . .                                  . . . 1 . . . .
 . . . 1 . . . .       pole magických čísel        . . . . . . . .
 . . . . . . . .                                  . 1 . . . . . .
 . 1 . P . . 1 .   *      rook_magic[P]       =    . . . . 1 . . .
 . . . 1 . . . .                                  . . . . . . . .
 . . . . . . . .                                  . . 1 . . . . .
 . . . . . . . .                                  . . . . . . 1 .
```

Vynásobením magickým číslem dostaneme index. Zeleně zabarvená část vypočítaného čísla je index. Teď už jenom stačí bitovým posunem dostat jenom posledních 10 bitů čísla a máme index.

3.2.1 Debugování

Problém s debugováním generátoru pohybů je ohromné množství možných pozicí, takže je velmi obtížné hledat všechny chyby v generátoru pouhým náhodným hraním. Proto byl vymyšlen perft (PERFormance Test)[8]. Perft projíždí všechny možné pozice do určité hloubky a konečné uzly stromu sečte. Potom výsledek porovnám se známými správnými výsledky.

Výsledek spočítáme klasickým DFS (Depht First Search). Budeme rekurzivně volat naší funkci na všechny pohyby a až dorazíme do hloubky jedna tak vrátímě počet všech pozic. Ty se potom v každé větvi sečtou. Nakonec dostaneme všechny konečné pozice. Tady je kód moji perft funkce.

```
unsigned long long perft(int depth, Board &board) {
    MoveGenerator gen = MoveGenerator();
    if (depth == 0) {
        return 1;
    }
    if (depth == 1) {
        return gen.getLegalMoves(board).size();
    }
    unsigned long long nodes = 0;
    for (auto move : gen.getLegalMoves(board)) {
        Board next_board = board;
        next_board.MakeMove(move);
        nodes += perft(depth - 1, next_board);
    }

    return nodes;
};
```

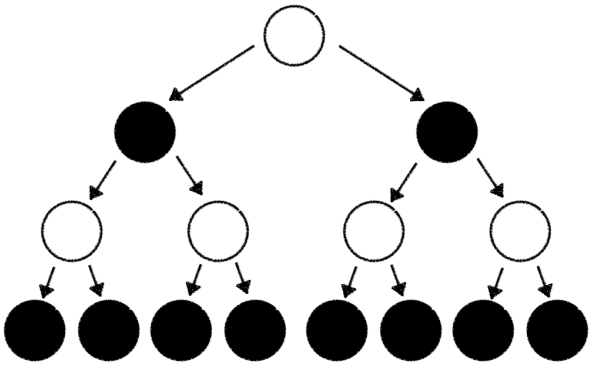
Tuto funkci potom využiji na ověření správnosti generátoru takto. Správné hodnoty konečných pozic jsem sebral z internetu.[9] Poté jsem jenom pomocí c++ knihovny Catch2 porovnal hodnoty.

```
    REQUIRE(perft(1, board) == 20);
    REQUIRE(perft(2, board) == 400);
    REQUIRE(perft(3, board) == 8902);
    REQUIRE(perft(4, board) == 197281);
    REQUIRE(perft(5, board) == 4865609);
    REQUIRE(perft(6, board) == 119060324);
```

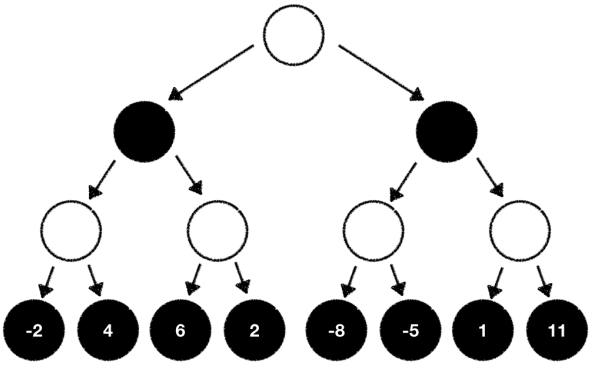
3.3 Prohledávání pozic

3.3.1 Minimax

Minimax algoritmus rekurzivně projde herní strom do nějaké hloubky a na posledním uzlu vrátí evaluaci pozice. Při vracení rekurze porovnám všechny potomky pozice a vyberu tu



Obrázek 3.3: Minimax strom. Barva uzlu určuje hráče na tahu v dané pozici



Obrázek 3.4: Minimax strom. Poslední pozice ohodnocené.

nejvýhodnější pro hráče právě na tahu.

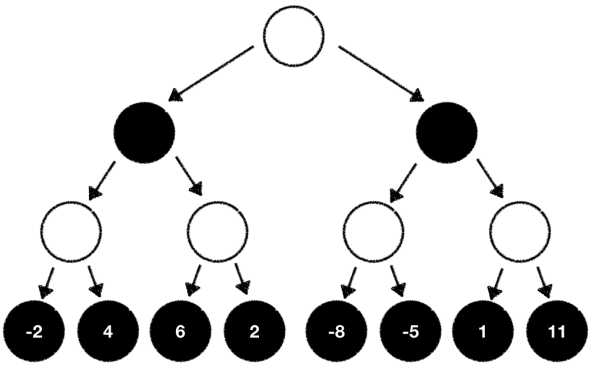
Takhle vypadá vygenerovaný minimax strom, který ještě nemá ohodnoceny poslední uzly. 3.3

Následně evaluační funkcí ohodnotím poslední uzly.3.4

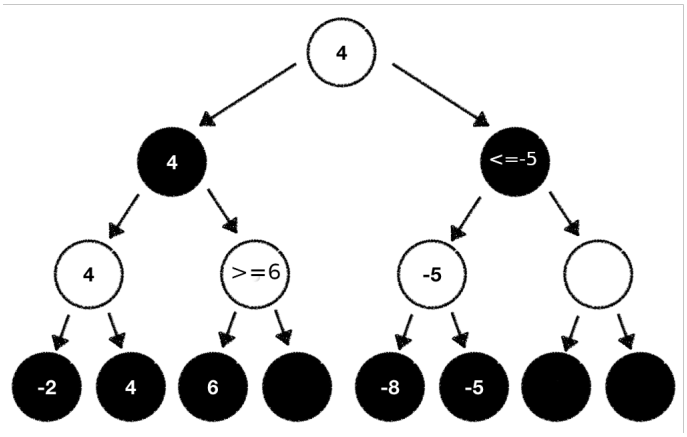
Teď půjdeme zespodu nahoru. Uzel porovná hodnoty všech potomků a nastaví svojí hodnotu na tu nejvýhodnější pro hráče na tahu. Podle toho poté vybereme nejlepší pohyb.3.5

3.3.2 Alpha-Beta pruning

Alpha-Beta pruning je vylepšení na algoritmu minimax. Jde o to, že není třeba vyhodnocovat všechny konečné uzly, protože při postupném vyhodnocování stromu zjistíme, že do některých větví se nikdy nemůžeme dostat za perfektního hraní nepřítele. Viz obr.3.6



Obrázek 3.5: Minimax strom. Všechny pozice ohodnocené

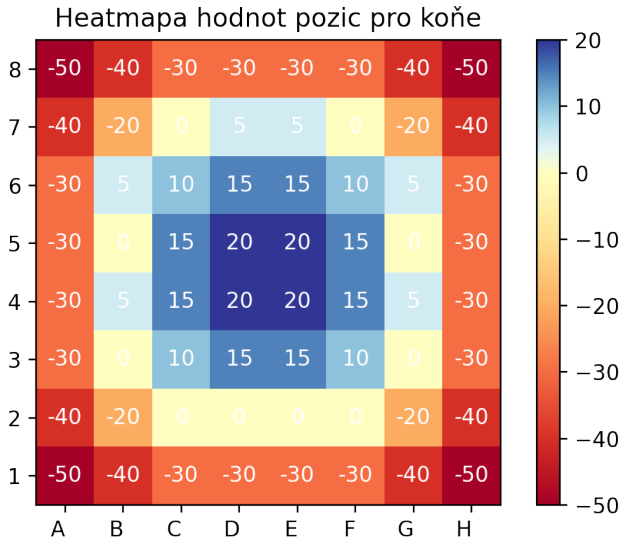


Obrázek 3.6: Ohodnocený strom pomocí alfa-beta pruningu. Prázdné uzly byli ořezány, jelikož je nebylo třeba prohledávat

Abychom odřízli co nejvíce takovýchto větví, je klíčové pořadí pohybů. Proto existují jsou různé techniky jak seřadit pohyby, aby nejvíce nadějné pohyby byli prozkoumány jako první. Například pohyb z transpoziční tabulky, sebrání nejcennější figurky nejméně cennou figurky a spoustu další heuristik pro dobré seřazení pohybů.[7]

3.4 Evaluace

Dobrá evaluační funkce je klíčová pro šachový engine, protože na základě ní rozhoduje jaké pozice jsou dobré. Evaluační funkce se může skládat z mnoha heuristik. Nejzákladnější heuristika je kolik jakých figurek má jaká strana. Na tuto heuristiku potřebujeme figurkám přidělit vhodnou hodnotu podle toho jak jsou cenné. Poté jenom sečteme celkovou hodnotu



Obrázek 3.7: Heatmapa hodnot pro koně, vytvořená pomocí knihovny matplotlib v pythonu.

bílých a černých figurek, odečteme je od sebe a změníme znamení podle toho kdo je na řadě.

Takhle vypadá útržek kódu.

```
int score = 0;
int side = board.on_turn ? -1 : 1;
for (int i = 0; i < 5; ++i) {
    int white = __builtin_popcountll(board.all_bitboards[WHITE][i]);
    int black = __builtin_popcountll(board.all_bitboards[BLACK][i]);
    score += (white - black) * opening_piece_values[i] * side;
}
```

Dále je dobré nějak ohodnotit kde jsou figurky postavené. Například jestli je kůň aktivní ve středu, nebo jestli je v podstatě k ničemu v rohu. Proto si vytvořím pole 64 čísel pro každý typ figurky. Pole bude obsahovat hodnoty, které budou určovat hodnotu dané figurky na daném místě. Pro koně vybadá toto pole takto. Viz obr.3.7 Poté takto ohodnotím kvalitu postavení všech figurek a přičtu to k finálnímu skóre pozice.

Pro účely mého jednoduchého šachového bohatě stačí tyto dvě heuristiky, ale používá se jich mnohem více. Například celková struktura pěšáků, či síla kombinace dvou střelců. Evaluační funkce taky může brát ohled na to v jaké fázi hry se hra nachází.

3.5 Transpoziční tabulka

Jelikož se často stává při prohledávání pozic, že narazíme na pozici, kterou už jsme někdy prohledávali, je výhodné si prohledané pozice ukládat. V c++ na to používám datovou strukturu unordered_map, což je v podstatě hashmapa, která má časovou složitost vkládání i čtení $O(1)$, díky čemuž je ideální na ukládání pozic.

Zobrist hašování

Jako klíč potřebuju vytvořit hash pozice, protože kdybychom použili všechny informace, museli bychom jako klíč použít 6 bitboardů plus informace o en passantu, což je příliš. Vzhledem k tomu, že potřebujeme hash pozice vytvářet velmi často, je třeba aby hashovací funkce byla hodně rychlá. Proto byla vytvořena metoda zobrist hashing[16]. Metoda se skládá ze dvou částí.

1. Předpočítání náhodných čísel a zpočítání prvního hashe.

   Pro každý bit na každém bitboardu vytvořím náhodné 64-bitové číslo. Poté se vytvořím 64-bitové číslo kde budu udržovat hash pozice. Následně projdu všechny bitboardy a pro každý obsazený bit vyXORuju příslušné náhodné číslo na mojí hash pozici. Tím dostanu hash počáteční pozice.

2. Aktualizování hash čísla podle zahraného pohybu.

   Krása zobrist hashování je, že když zpočítáme počáteční hash, můžeme poté hash pouze aktualizovat cca dvěma XOR operacemi. Když provedeme nějaký pohyb, potřebujeme z hashe odstranit figurku která se pohla a přidat figurku tam kam se pohla. To uděláme tak, že použijeme náhodné hodnoty z minulého kroku. Abychom figurku odstranili z hashe, stačí nám na náš hash znovu vyXORovat náhodné číslo, které koresponduje s touto figurkou na této pozici. Potom do hashe přidáme náhodné číslo, které koresponduje s pozicí kam jsme figurku přesunuli.

Proč to funguje?

Tato metoda funguje díky dvoum atributům XOR operace. Zaprvé nám pomáhá asociativita XOR operace, tzn. že nezáleží na pořadí operací, což je přesně to co potřebujeme aby

pozice měla stejný hash pokud se tam dostaneme různou variací pohybů. Zadruhé nám pomáhá to, že XOR dvou stejným čísel je nula. Takže když chceme z hash čísla odstranit nějakou figurku, tak pouze na náš hash vyXORujeme náhodné číslo korespondující s pozicí figurky.

3.6 Databáze otevíracích pohybů

Počáteční fáze šachů je v dnešní době prozkoumána do velké hloubky, takže naše znalosti mnohem převyšují schopnosti šachového najít nejlepší pohyby. Proto je vhodné mít nějakou databázi pohybů, kterou použijeme na začátku hry. Abych si vytvořil svojí databázi musím stáhnout nějakou sadu her. Vybral jsem si sadu her MillionBase 2.5[14], což je databáze 2.5 milionu kvalitních šachových partií ve formátu PGN[3].

3.6.1 PGN formát

PGN je standardní formát na zaznamenání her v šachách. Je relativně snadno čitelný pro lidi, ale není tak snadné ho počítačově zparsovat, protože není možné vytvořit pohyb bez znalosti stavu hry. Jde o to, že v pgn formátu se používá co nejméně informací na identifikaci figurky kterou chceme hýbat. Takže například pokud chceme v PGN formátu zapsat, že chceme pohnout koněm na určité místo, a máme koně pouze jednoto, tak zapíšeme jenom že hýbáme koněm a kam s ním hýbáme. To, kterým koňem a odkud hýbáme vyplívá z toho, že máme koňe pouze jednoho. Pokud máme dva koně, tak bychom k tomu museli ještě připsat v jakém sloupci, či řádku se nachází. Takhle vypadá ukázková hra v PGN formátu.

```
[Event "Spring "]
[Site "Budapest open"]
[Date "1996.??.??"]
[Round "1"]
[White "Aadrians, M. (wh)"]
[Black "Dekic, J. (bl)"]
[Result "0-1"]
[BlackElo "2320"]
[ECO "D82"]
```

```
1. d4 Nf6 2. c4 g6 3. Nc3 d5 4. Bf4 Bg7 5. Be5 dxc4 6. e3 Nc6 7. Qa4 O-O
8. Bxf6 Bxf6 9. Bxc4 a6 10. Bd5 b5 11. Qd1 Bb7 12. a3 e6 13. Bf3 Na5
14. Bxb7 Nxb7 15. b4 c5 16. bxc5 Nxc5 17. Nf3 Qa5 18. Qc2 Na4 19. Rc1 Rac8
20. O-O Qxc3 21. Qe2 Qxa3 22. Rc2 Rxc2 23. Qxc2 Qc3 24. Qe4 Rc8 25. g3 Qc2
26. Qb7 Qc6 0-1
```

Svůj PGN parser[1] chci mít co nejjednodušší, takže budu ignorovat informační tagy[2] a budu prohledávat maximálně do hloubky 14 pohybů. Abych nemusel pokaždé před začátkem programu parsovat 2.5 miliónu her, napsal jsem si svojí takovou databázi a budu tedy vysledky ukladat do souborů. Nejdříve si načtu všechny pohybu do datové struktury map<bitboard, unordered_map<string, int>> , což je seřazená hash mapa podle klíče, která obsahuje neseřazenou hashmapu pohybů s jejich četnostmi. Abych efektivně tuto datovou strukturu využil i ze souboru, vytvořil jsem si datový a indexový soubor. Indexový soubor obsahuje seřazené hash pozice a index pohybů pro určitý hash v datovém souboru. Datový soubor obsahuje na každém řadku všechny pohybý které byli v dané pozic zahrány a jejich četnost.

Když budu chtít vyhledat zahrané tahy pro určitou pozici. Vyhledám pomocí binárního vyhledávání hash v indexovém souboru. Tím dostanu pozici pohybů v datovém souboru a podle této pozice tyto pohyby načtu. Časová složitost vyhledávání bude

$$O(\log n + m)$$

kde n je počet hash pozic uložených v indexovém souboru a m je počet různých zahraných tahů v určité pozici.

Tady je útržek kódu z mého parseru.

```
1 depth++
2 database[board.zobrist_hash][token]++;
3 board.MakeMoveFromPGN(token);
```

Je to kód ze smyčky která prochází už pohyby pgn formátu, které jsou uložené jako string ve proměnné tok. Nejdřív zvětším hloubku prohledávání o jednu. Potom přidám jedničku ke četnosti pohybu, který se chystám zahrát v dalším příkazu.

[1]Parser je program na překlad určitého vstupu do žádaného výstupu. V mém případě chci přetvořit PGN formát do struktury kterou používám na ukládání pohybů

[2]Nad každou hrou. Obsahují metadata o hře. Například ELO hráčů, či místo konáni hry.

Kapitola 4

GUI

Pro vytvoření GUI[1] mého programu jsem zvolil OLC::PixelGameEngine[2], protože už jsem v něm pracoval, je minimalistický a rychlý. Vzhledem k tomu, že můj hlavní cíl mojí práce není GUI a bylo by časově náročné dělat nějaké propracované GUI, tak jsem ho vytvořil pouze na hraní za bílého a bez jakéhokoliv menu. V PixelGameEnginu neexistují žádné běžné GUI prvky, pouze můžu kreslit tvary, obrázky a samotné pixely. Vždycky, když se něco změní na hrací ploše, tak ji vykreslím celou znova. Jediné co dělám je, že při kliku na figurku zvýrazním všechny možné tahy s touto figurkou a při kliku na nějaké místo tou figurkou táhnu. Když zahraje tah člověk, začne s počítáním tahu počítač. Přibližné prvních 5 bude velmi rychlých, protože jsou zahrány z databáze tahů. Poté je na to engine sám a musí používat vyhledávací funkci. Je důležité aby GUI beželo v jiném vlákně než vyhledávací funkce, protože jinak by GUI přestalo reagovat. Mám jednu atomic[2] proměnnou, která rozhoduje v jaké fázi je program, tedy jestli je na řadě bot, či hráč. Pro ilustraci přikládám screenshot GUI. Viz obr.4.1

Je z něj teda patrné, že jsem si dělal grafiku sám.

4.1 Návod na použití

Pro zkompilování a použití mého enginu s GUI budete potřebovat počítač s linuxem.

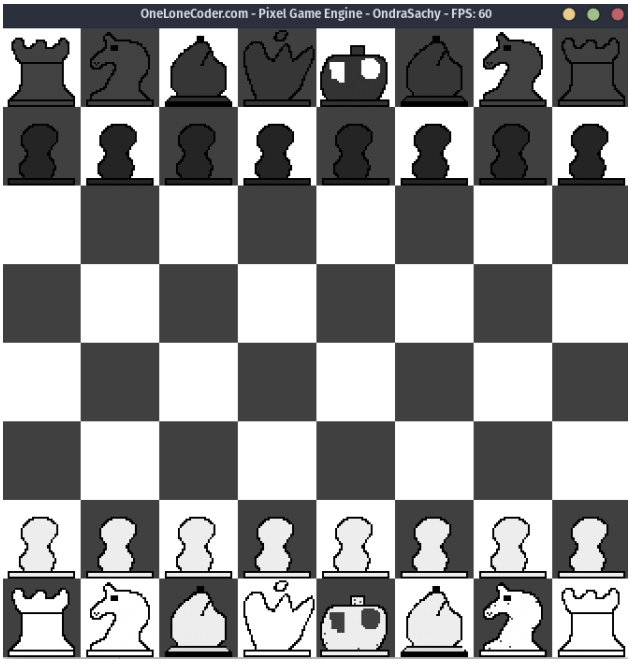
- Nainstalujte si tyto dependence, které obsahují kompilátor a knihovny potřebné pro GUI

  ```
  sudo apt install build-essential libglu1-mesa-dev libpng-dev
  ```

- Poté se dostaneme do adresáře src a zkompilujeme projekt.

[1] Graphical User Interface

[2] atomic proměnná v c++ znamená, že k ní nemůžou dvě vlákna přistoupit zároveň



Obrázek 4.1: Mnou vytvořený design GUI

```
make
```

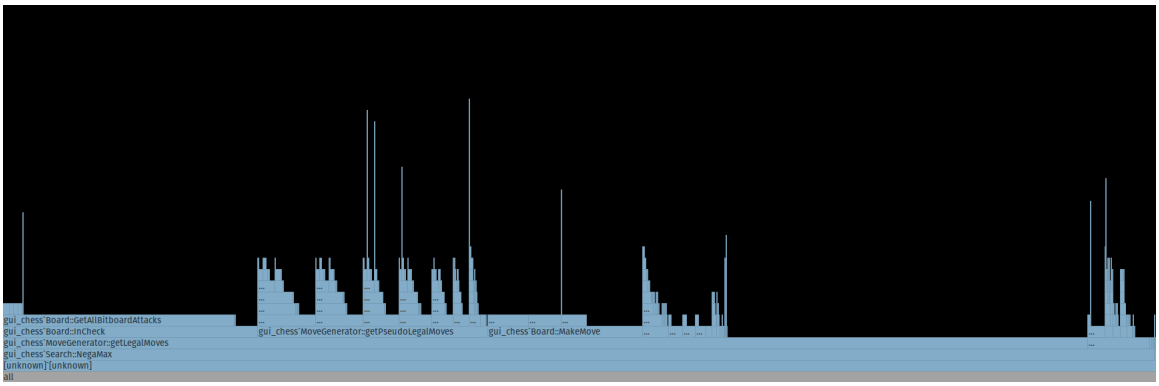
-

Kapitola 5

Optimalizování

Klíčový pro šachový je rychlost. Proto abychom měli šachový engine co nejrychlejší musíme ho různými způsoby optimalizovat. Například použití rychlejšího algoritmu, efektivnější datové struktury, či prosté odstranění redundantních operacích. Je důležité primárně optimalizovat funkce, které jsou nejvíce časově náročné, abychom z našeho úsilí vytěžili co nejvíce. Pro to využiju tzv. profiler. Jedná se o program, který analyzuje chod programu a umožní mi zjistik kolik jaké funkce zabírají času. Já rád používám linuxový program perf[1] s visualizací v podobě plamenového grafu[4]. Viz obr. 5.1 Každá funkce je tam v podobně horizontální čáry. Čím je čára delší, tím více procent zabírá z celkového běhu programu. Pokud je nějaká čára nad jinou čarou, znamená to, že je funkce volána ve funkci pod ní. Z grafu 5.1 je vidět, že nejvíc času zabíra funkce Negamax (minimax vyhledávání) a většinu této funkce zabírá zanořená funkce GetLegalMoves (Vrací možné legální pohyby). Takže až budu chtít ještě více optimalizovat, tak to bude generování pohybů. Můj šachový engine zvládne pomocí Perft (viz kapitola 3.2.1) prohledat 4865609 pozic za přibližně jednu sekundu. Pro porovnání, nejlepší volně dostupný šachový engine na světě, Stockfish stejný počet pozic prohledal za 58 milisekund, takže je ještě spousta prostoru kam můžu svůj engine zlepšovat.

[1] obsažený v linuxovém kernel



Obrázek 5.1: Plamenový graf mého šachového enginu.

Kapitola 6

Závěr

Cílem mojí maturitní práce bylo vytvořit šachový engine a popsat během.

Seznam tabulek

Seznam obrázků

Literatura

[1] Claude, E. S.: Programming a Computer for Playing Chess. *Philosophical Magazine, Ser*, ročník 7, č. 41, 1950: str. 314.

[2] David Barr: olc::PixelGameEngine. Dostupné na: `OneLoneCoder.com`.

[3] Edwards, S. J.: Portable game notation specification and implementation guide. *Retrieved April*, ročník 4, 1994: str. 2011.

[4] Gregg, B.: Dostupné na: `http://www.brendangregg.com/flamegraphs.html`, 2021.

[5] Přispěvatelé Chess Programming Wiki: Alpha-Beta. Dostupné na: `https://www.chessprogramming.org/Alpha-Beta`, 2021.

[6] Přispěvatelé Chess Programming Wiki: Bitboards. Dostupné na: `https://www.chessprogramming.org/Bitboards`, 2021.

[7] Přispěvatelé Chess Programming Wiki: Move Ordering. Dostupné na: `https://www.chessprogramming.org/Move_Ordering`, 2021.

[8] Přispěvatelé Chess Programming Wiki: Perft. Dostupné na: `https://www.chessprogramming.org/Perft`, 2021.

[9] Přispěvatelé Chess Programming Wiki: Perft results. Dostupné na: `https://www.chessprogramming.org/Perft_Results`, 2021.

[10] Přispěvatelé Wikipedie: Game tree. Dostupné na: `https://en.wikipedia.org/w/index.php?title=Game_tree&oldid=1003093164`, 2021.

[11] Roeder, O.: Chess's New Best Player Is A Fearless, Swashbuckling Algorithm. Dostupné na `https://fivethirtyeight.com/features/chesss-new-best-player-is-a-fearless-swashbuckling-algorithm/`, 1 2018.

[12] Rustad-Elliott, R.: Fast Chess Move Generation With Magic Bitboards. Dostupné na https://rhysre.net/fast-chess-move-generation-with-magic-bitboards.html.

[13] Schaeffer, J.; Björnsson, Y.; Kishimoto, A.; aj.: Checkers Is Solved. *Science*, ročník 317, 10 2007: s. 1518–1522.

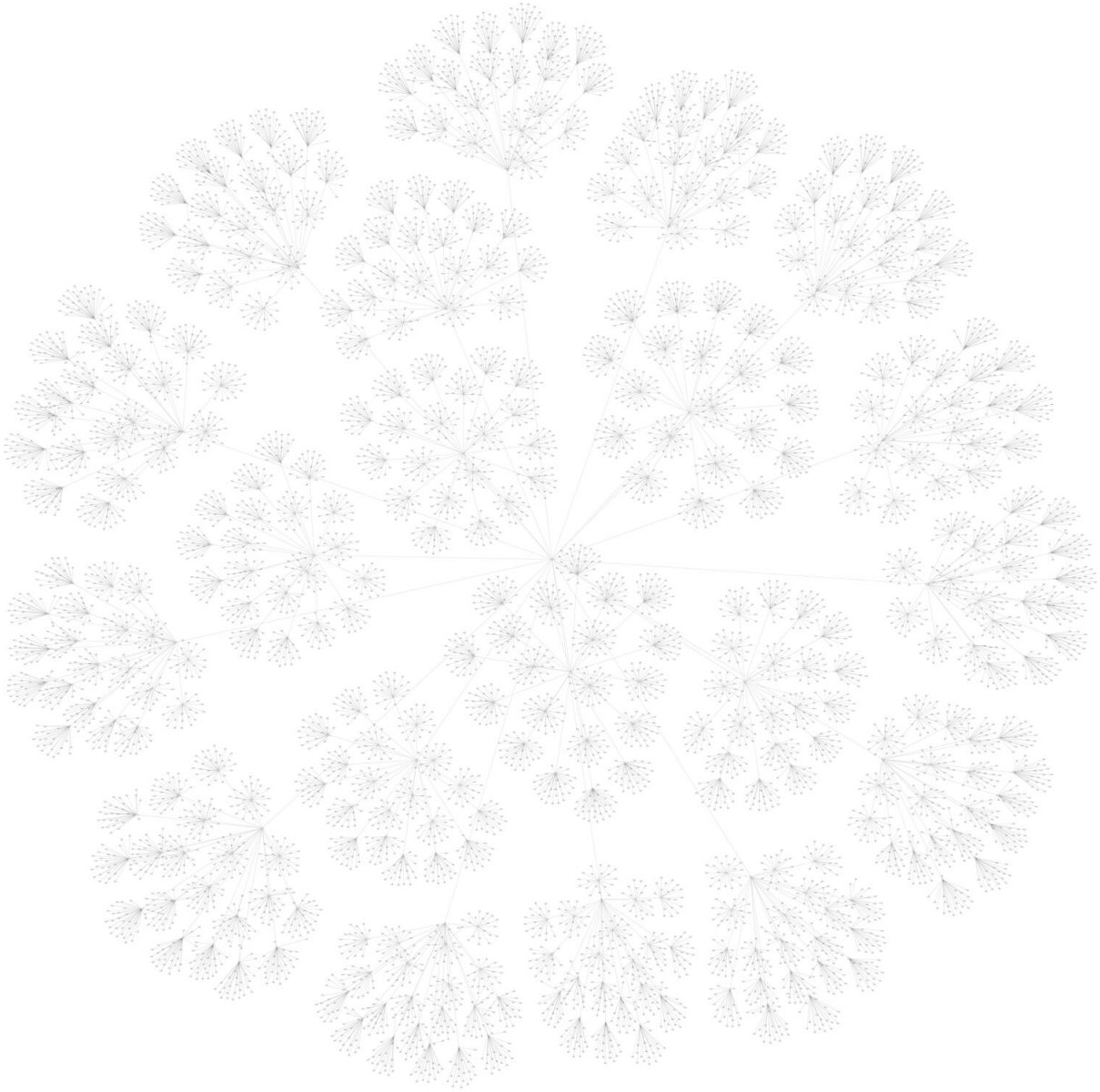
[14] Schroder, E.: Perft results. Dostupné na: http://rebel13.nl/download/data.html, 2021.

[15] Yang, D.: Introducing NNUE Evaluation. Dostupné na https://stockfishchess.org/blog/2020/introducing-nnue-evaluation/.

[16] Zobrist, A. L.: A new hashing method with application for game playing. *ICGA Journal*, ročník 13, č. 2, 1990: s. 69–73.

Příloha A

Přílohy



Obrázek A.1: větvění šachů po třech pohyběch