



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

18-20-M/01 Informační technologie

## 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 mnou vytvořený šachový engine a vysvětluje některé techniky a algoritmy, které šachovní 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; Šachový stroj; Teorie her; Minimax; Bitboard

## Annotation

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

**Keywords:** Chess; Chess bot; Chess engine; Game theory; Minimax; Bitboard

## Poděkování

Chtěl bych poděkovat mým kamarádům Tomáši Zemanovi, Milanu Jiříčkovi, Danielu Maškovi a Richardu Kropáčkovi za gramatickou korekturu.

# Obsah

<b>1</b>	<b>Úvod</b>	<b>8</b>
<b>2</b>	<b>Popis a struktura šachového enginu</b>	<b>9</b>
2.1	Teorie her . . . . .	9
2.2	Dělení šachových enginů . . . . .	10
2.3	Struktura šachového enginu . . . . .	11
2.4	Historie Šachových enginů . . . . .	12
<b>3</b>	<b>Implementace</b>	<b>13</b>
3.1	Bitboardy . . . . .	13
3.2	Generování pohybů . . . . .	14
3.2.1	Debugování . . . . .	17
3.3	Prohledávání pozic . . . . .	19
3.3.1	Minimax . . . . .	19
3.3.2	Alpha-Beta pruning . . . . .	20
3.4	Evaluace . . . . .	21
3.5	Transpoziční tabulka . . . . .	22
3.6	Databáze otevíracích pohybů . . . . .	23
3.6.1	PGN formát . . . . .	24
<b>4</b>	<b>GUI</b>	<b>26</b>
4.1	Návod na použití . . . . .	27
<b>5</b>	<b>Optimalizování</b>	<b>28</b>
<b>6</b>	<b>Závěr</b>	<b>29</b>
	<b>Přílohy</b>	<b>32</b>
<b>A</b>	<b>Přílohy</b>	<b>33</b>

# Kapitola 1

## Úvod

V této maturitní práci popíšu, jak byl vytvořen můj šachový engine. Budu se snažit popsat různé techniky a algoritmy, které šachové enginy využívají. Celou práci budu psát v C++, a používat specifické funkce pro kompilátor GCC. Většině popsaným algoritmům bude určena časovou složitost. Na konec bude vložen návod na zkompilování a používání enginu.

Pro implementaci šachových pravidel jsem si pro simplicitu vybral verzi šachů bez rošády. Nebudu popisovat základní datové struktury a algoritmy, jako je binární vyhledávání nebo hashmapy.



# Kapitola 2

## Popis a struktura šachového enginu

### 2.1 Teorie her

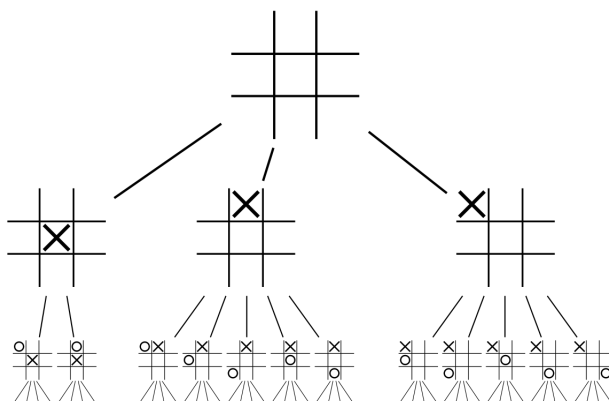
Šachy jsou stále nevyřešená hra, tzn. neví se, jak vypadá bezchybná hra. Jsou nevyřešené, protože počet možných pozic je enormní. Už jenom po třech pohybech je 8 902 možných variant hry. Pro představu tady je vygenerovaný graf pomocí programu Graphviz, který ukazuje, jak se hra po třech tazích větví. Viz obr. A.1. Obrázek v plné velikosti je přiložen v repozitáři.

Průměrný počet pohybů ve všech situacích je přibližně 35. Průměrná hra má 80 pohybů, takže když chceme dostat hodně nepřesný odhad možných šachových 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 je v nejbližší době, či dokonce nikdy nevyřešíme. Pro zajímavost dáma má přibližně  $5 \times 10^{20}$  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. Na základě těchto pravděpodobností se následně bude 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



Obrázek 2.1: Herní strom piškvorek [10]

## 2.2 Dělení šachových enginů

Zjednodušeně se dají šachové enginy rozdělit do dvou skupin.

- 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 doháněny prohledáváním velkého množství pozic.
  - Detaily budou v kapitole 3.3
- Šachový bot založený na neuronových sítích a prohledávání stromu metodou Monte Carlo
  - Hodnocení pozic provádí neuronové sítě, které byly natrénovány 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 úspěch.

## 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<sup>1</sup>, otestuje pohyby a vyřadí ty, u kterých je vlastní král v šachu.
- Třída na reprezentaci stavu hry
  - Udrží informace o pozicích, právech figurek a případně s nimi hýbe.
  - Na udržování pozic se většinou používají bitboardy<sup>2</sup>. 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 3.5
- Evaluace pozic
  - Hodnotí pozici podle hodnoty, umístění a struktury figurek.
  - Je klíčová pro chování šachového enginu, protože na základě ní se rozhoduje v prohledávání pozic.

---

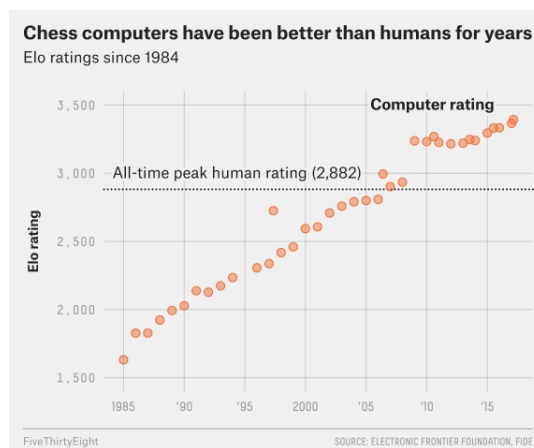
<sup>1</sup>Pohyby, které odpovídají tomu, jak se mají figurky hýbat, ale je u nich možnost, že by dostali svého krále do šachu.

<sup>2</sup>Bitboard nebo bitmapa je 64 bitové číslo, kde v jeho binární podobě každá jednička znamená zabranou pozici na šachovém poli.

- Detaily v kapitole 3.4
- Třída na prohledávání pohybů
  - Prohledává strom hry a vrací nejlepší nalezený pohyb.
  - Používá různé techniky, jak prohledávat co nejnadějnější větve a odřezává větve, u kterých není šance na úspěch.
  - Detaily v kapitole 3.3

## 2.4 Historie Šachových enginů

První plně funkční šachové enginy se začaly objevovat v 60. letech minulého století. Algoritmy pro šachové enginy existovali už dříve, ale jejich použití bylo bržděno tehdejším hardwarem. Už v 50. letech minulého století se znaly všechny algoritmy, aby mohl být stvořen použitelný šachový engine. Síla šachových enginů poté prudce rostla. Zaprvé kvůli Mooreově zákonu<sup>3</sup> a zadruhé díky zlepšování šachových algoritmů. V obr. 2.2 vidíme vývoj Elo<sup>4</sup> 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 neuronové sítě a při vydání této změny získal více než 80 Elo bodů.[16]



Obrázek 2.2: Vývoj Elo šachových enginů[11]

<sup>3</sup>Vypozorovaný zákon, že každé dva roky se zdvojnásobí hustota tranzistorů v integrovaném obvodu.

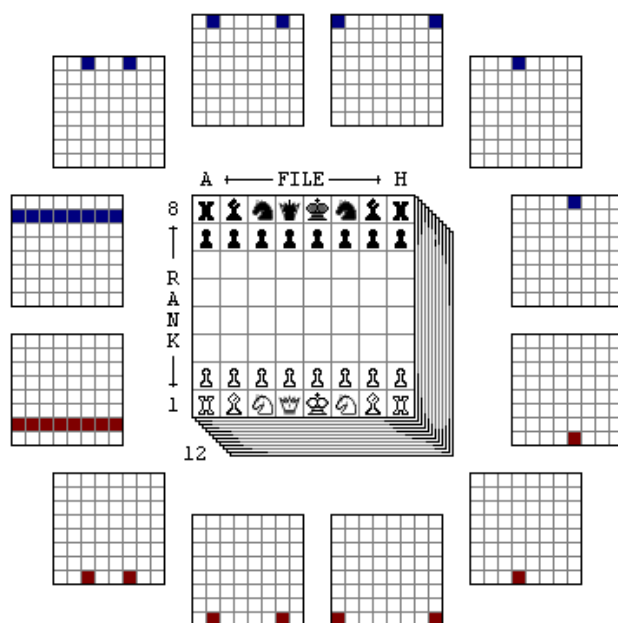
<sup>4</sup>Statistické ohodnocení výkonnosti hráče

# 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 bylo možné 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. Na bitboardy také lze používat logické operace, takže můžeme například všechny pěšáky posunout o 8 bitů doprava (pohyb nahoru) v jednom clock cyclu. V C++ byla na reprezentaci bitboardu použita datová struktura unsigned long long.



Obrázek 3.1: Reprezentace herní plochy pomocí bitboardů[6]

V kódu jsou bitboardy udržovány ve třídě Board. V této třídě jsou kromě bitboardů také metody, které určité bitboardy spojí logickou operací OR. Takhle vypadá definice třídy.

---

```
1 typedef unsigned long long bitboard;
2 class Board {
3     public:
4         //vsechny bitboardy
5         //6 pro kazdou figurky a 2 pro kazdou barvu
6         bitboard all_bitboards[2][6]{};
7
8         // spojené pozice
9
10        // vraci bitboard figurek určité barvy
11        bitboard PiecesOfColor(bool color);
12
13        // vraci bitboard vseh Nigurek
14        bitboard AllPieces();
15 }
```

---

## 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, kde se figurky nachází. Když potom narazíme na figurku, které chceme vypočítat možné pohyby, stačí vzít předpočítaný bitboard. Generování potom probíhá postupným procházením bitů předpočítaného bitboardu. Aby bylo procházení bitboardu, co nejrychlejší je použita funkce `__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, kterým je jednička. Tím dostaneme index pozice, kam může figurka jít. Prohledaný bit poté nastavíme na nulu, abych přístě dostal další jedničku.

Na ukázkou je přiložen úryvek kódu pro generování pohybů koněm.

---

```
1 bitboard enemy_pieces = board.EnemyPieces();
2 bitboard my_pieces = board.MyPieces();
```

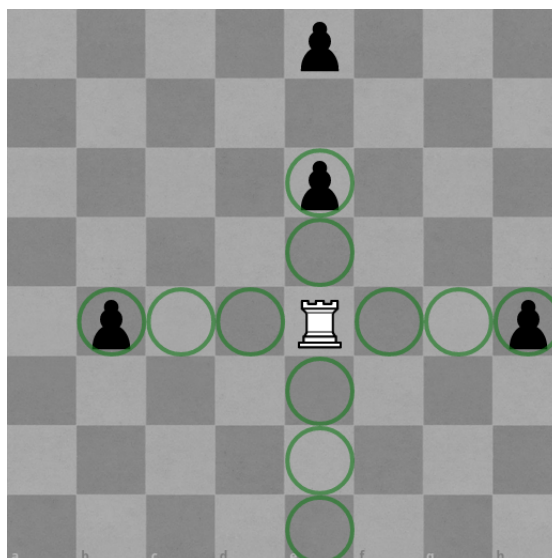
```

3      bitboard my_knights = board.all_bitboards[board.on_turn][KNIGHT];
4      int from;
5      int to;
6      while (my_knights) {
7          // smyčka bude procházet bitboard my_knights, dokud bitboard nebude prázdný
8
9          // nactení pozice dalšího kone
10         from = __builtin_ffsll(my_knights);
11         from--;
12         // smazání nacteného kone z proměnné my_knights
13         my_knights ^= 1ULL << (from);
14         // útoky ze současného kone
15         bitboard attacks = precomp.precomputed_knights[from];
16         // odstranění pohybu, které by skončilo na nějaké me figurce
17         attacks &= ~my_pieces;
18         while (attacks) {
19             //nactení pozice pro pohyb
20             to = __builtin_ffsll(attacks);
21             to--;
22
23             // smazání nactené pozice
24             attacks ^= 1ULL << (to);
25             if (enemy_pieces >> to & 1ULL) {
26                 // pohyb sebere figurku
27                 moves.push_back(Move{from, to, KNIGHT, board.getPieceAt(to), true});
28             }
29             else {
30                 // pohyb nesebere figurku
31                 moves.push_back(Move{from, to, KNIGHT});
32             }
33         }

```

---

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



Obrázek 3.2: Možnosti pohybu věže

Vidíme, že nás tedy zajímá, jaké figurky blokují pohyb věže. Můžeme si předpočítat všechny možnosti, kde můžou být. Zajímají nás pouze figurky nacházející se na místech, kam se útočící figurka může hýbat. V tomto konkrétním případě je to 14 pozic, kde můžou být figurky umístěny. To je  $2^{14} = 16384$  pozic, jak můžou být figurky uspořádány. Toto číslo můžeme ještě zmenšit, protože na úplně krajních pozicích nezáleží. Krajní pozice totiž nic neblokují. Takže v tomto případě to je  $2^{10} = 1024$  pozic, což v dnešní době není problém udržet v paměti. 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. Lze je touto rychlou metodou vyjmout z bitboardu a indexovat pouze podle těchto určitých pozic. Funguje to tak, že se určitým číslem (magickým) vynásobí bitboard a to číslo je tak šikovné, že nám posune chtěné pozice na prvních 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. Použitá čísla nejsou má, ale jsou použita čísla, která již někdo spočítal.[12]

V tomto příkladu je ukázáno, jak by se vytvořil index pro tuto situaci.



bitboard bitů, které nás zajímají		bitboard obsazených pozic		bitboard 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 posledních 10 bitů čísla a máme index.

### 3.2.1 Debugování

Problémem u debugování generátoru pohybů, je ohromné množství možných pozic, 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 porovná se známými správnými výsledky.

Výsledek spočítáme pomocí klasického DFS (Depth First Search). Budeme rekurzivně volat naši funkci na všechny možné pohyby, a až dorazíme do hloubky jedna, tak vrátíme

počet všech pozic. Ty se potom v každé větvi sečtou. Nakonec dostaneme všechny konečné pozice. Tady je kód perft funkce.

---

```
1 unsigned long long perft(int depth, Board &board) {
2     MoveGenerator gen = MoveGenerator();
3     if (depth == 0) {
4         return 1;
5     }
6     if (depth == 1) {
7         // vrati pocet legalnich pohybu na konci vetve
8         return gen.getLegalMoves(board).size();
9     }
10    unsigned long long nodes = 0;
11
12    for (auto move : gen.getLegalMoves(board)) {
13        Board next_board = board;
14        // zahraje pohyb dalsiho uzlu
15        next_board.MakeMove(move);
16
17        // pricte pocet konecnych uzlu teto vetve
18        nodes += perft(depth - 1, next_board);
19    }
20    return nodes;
21 };
```

---

Správné hodnoty konečných pozic jsou použity z internetu.[9] Poté jsou pomocí C++ knihovny Catch2 porovnány s vypočítanými hodnotami. Perft funkci potom využijeme na ověření správnosti generátoru takto.

---

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

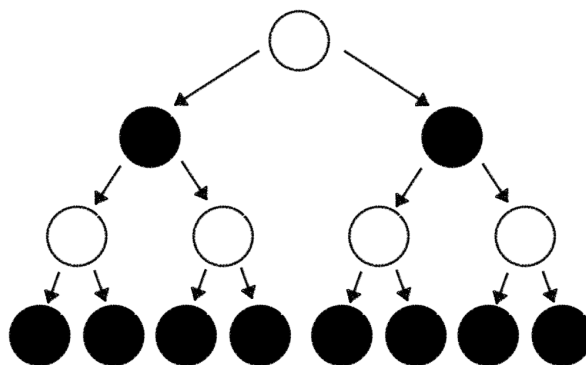
---

## 3.3 Prohledávání pozic

### 3.3.1 Minimax

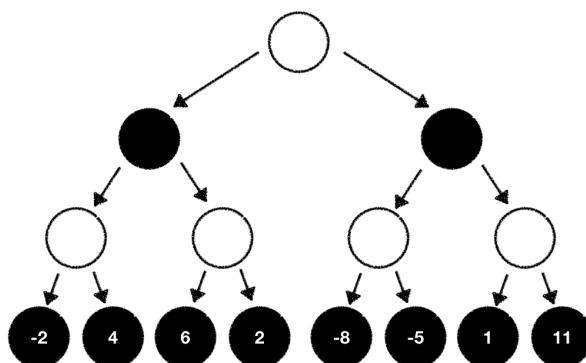
Minimax algoritmus rekurzivně projde herní strom do nějaké hloubky a na posledním uzlu vrátí evaluaci pozice. Při vracení rekurze porovná všechny potomky pozice a vyberu tu nejvýhodnější pro hráče, který je právě na tahu.

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



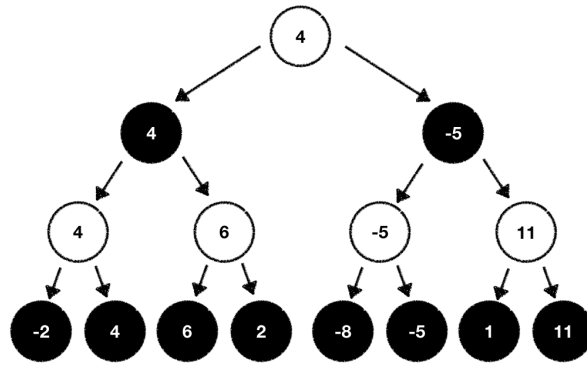
Obrázek 3.3: Minimax strom. Barva uzlu určuje hráče na tahu v dané pozici

Následně evaluační funkcí ohodnotí poslední uzly. 3.4



Obrázek 3.4: Minimax strom. Poslední pozice ohodnocené.

Ted' 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



Obrázek 3.5: Minimax strom. Všechny pozice ohodnocené

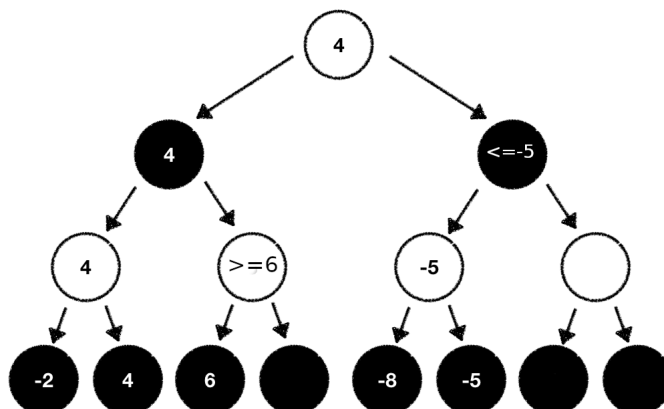
Časová složitost toho algoritmu je

$$O(b^d)$$

kde  $b$  je průměrný počet možných tahů a  $d$  je hloubka stromu.

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



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

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

Takhle vypadá útržek kódu.

---

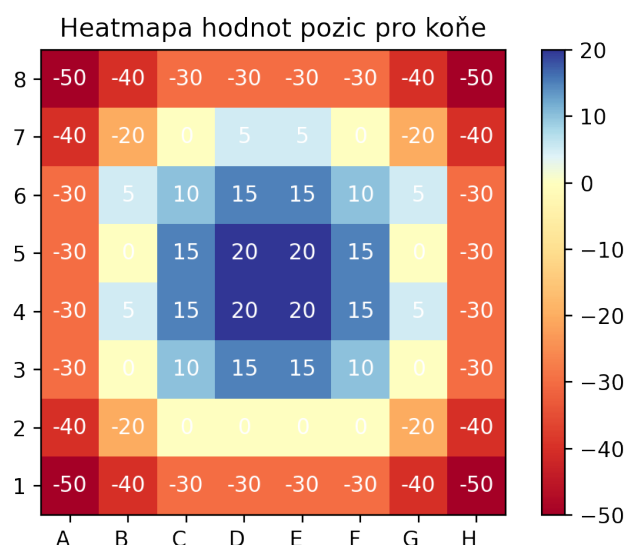
```

1  int score = 0;
2  int side = board.on_turn ? -1 : 1;
3  for (int i = 0; i < 5; ++i) {
4      int white = __builtin_popcount11(board.all_bitboards[WHITE][i]);
5      int black = __builtin_popcount11(board.all_bitboards[BLACK][i]);
6      score += (white - black) * opening_piece_values[i] * side;
7  }

```

---

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 vytvoříme pole 64 čísel pro každý typ figurky. Pole bude obsahovat čísla, která budou určovat hodnotu dané figurky na daném místě. Pro koně vybadá toto pole takto. Viz obr. 3.7. Poté ohodnotím kvalitu postavení všech figurek a přičtu to k finálnímu skóre pozice.



Obrázek 3.7: Heatmapa hodnot pozic pro koňe, vytvořená pomocí knihovny matplotlib v Pythonu.

Pro účely jednoduchého šachového bota bohatě stačí tyto dvě heuristiky, ale používá se jich mnohem více. Například celková struktura pěšáků či mobilita figurek. Evaluační funkce taky může brát ohled na to, v jaké fázi se hra nachází.

### 3.5 Transpoziční tabulka

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

#### Zobrist hašování

Jako klíč potřebujeme 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. [17] Metoda se skládá ze dvou částí.

### 1. Předpočítání náhodných čísel a spočítání prvního hashe.

Pro každý bit na všech bitboardech vytvoříme náhodné 64-bitové číslo. Poté vytvoříme 64-bitové číslo, kde budeme udržovat hash současné pozice. Následně projdu všechny bitboardy a pro každý obsazený bit vyXORuju příslušné náhodné číslo na její hash pozici. Tím dostanu hash počáteční pozice.

### 2. Aktualizování hash čísla podle zahraného pohybu.

Krása zobrist hashování je, že když spočítáme 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 figurkou na této pozici. Potom do hashe přidáme náhodné číslo, které koresponduje s pozicí, na kterou jsme figurku přesunuli.

## Proč to funguje?

Tato metoda funguje díky dvěma 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 stejných 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 enginu najít nejlepší pohyby. Proto je vhodné mít nějakou databázi pohybů, kterou použijeme na začátku hry. Abychom si vytvořili svojí databázi, musíme stáhnout nějakou sadu her. Zvolíme například 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 pohyb koněm na určité místo, a máme koně pouze jednoho, tak zapíšeme jenom že hýbáme koněm a kam s ním hýbáme. To, kterým koněm a odkud hýbáme vyplývá z toho, že máme koně pouze jednoho. Pokud máme koně dva, 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 0-0
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. 0-0 Qxc3 21. Qe2 Qxa3 22. Rc2 Rxc2 23. Qxc2 Qc3 24. Qe4 Rc8 25. g3 Qc2
26. Qb7 Qc6 0-1
```

---

PGN parser<sup>1</sup> vytvoříme co nejjednodušší, takže budeme ignorovat informační tagy<sup>2</sup> a budeme prohledávat maximálně do hloubky 14 pohybů. Abychom nemuseli pokaždé před začátkem programu parsovat 2.5 miliónu her. Napíšeme si vlastní key-value databázi<sup>3</sup>, která bude fungovat ze souboru na disku. Nejdříve si načteme všechny pohyby do datové struktury `map<bitboard, unordered_map<string, int>>`, což je seřazená hash mapa podle klíče, která obsahuje neseřazenou hashmapu pohybů s jejich četnostmi. Abychom

---

<sup>1</sup>Parser je program na překlad určitého vstupu do žádaného výstupu. V našem případě chceme přetvořit PGN formát do struktury, kterou používáme na ukládání pohybů.

<sup>2</sup>Jsou nad každou hrou. Obsahují metadata o hře. Například ELO hráčů či místo konání hry.

<sup>3</sup>Je to v podstatě hashmap, akortá je uložena na disku. Pomocí klíče (hashe) můžeme dostat hodnotu (optimální tahy)



tuto datovou strukturu využili efektivně 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 řádku všechny pohyby, které byly v dané pozici zahrány a jejich četnosti.

Když budu chtít vyhledat zahrané tahy pro určitou pozici, vyhledám pomocí binárního vyhledávání hash v indexovém souboru. Tím dostanu pozici pohybů v datovém souboru, podle které 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 zahráných tahů v určité pozici.

Tady je útržek kódu z parseru.

---

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

---

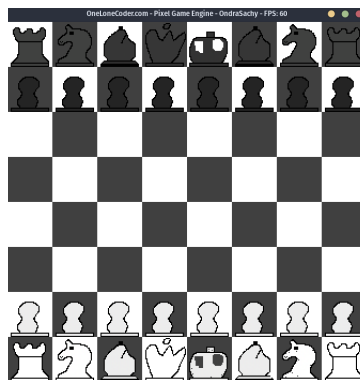
Je to kód ze smyčky, která prochází pohyby pgn formátu, které jsou uloženy jako string v proměnné token. Nejdříve zvětšíme hloubku prohledávání o jedna. Potom přidáme jedničku k četnosti pohybu, který se chystáme zahrát v dalším příkazu.

# Kapitola 4

## GUI

Pro vytvoření GUI<sup>1</sup> mého programu byl zvolen OLC::PixelGameEngine[2], protože jsem v něm již pracoval. Je minimalistický a rychlý. Vzhledem k tomu, že hlavním cílem této práce není GUI a bylo by časově náročné dělat propracované GUI, tak bylo vytvořeno pouze na hraní za bílého a bez jakéhokoliv menu. V PixelGameEnginu neexistují žádné běžné GUI prvky, pouze můžeme kreslit tvary, obrázky a samotné pixely. Pokaždé, když se něco změní na hrací ploše, tak ji vykreslíme celou znova. Jediné co děláme je, že při kliku na figurku zvýrazníme všechny možné tahy s touto figurkou a při kliku na nějaké místo touto figurkou táhnu. Když zahraje tah člověk, začne s počítáním tahu počítač.

Přibližně 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 běželo v jiném vlákne, než vyhledávací funkce, protože jinak by GUI přestalo reagovat. Máme jednu atomic<sup>2</sup> proměnnou, která rozhoduje, v jaké je program fázi, tedy jestli je na řadě bot či hráč. Pro ilustraci je tady přiložený screenshot GUI. Viz obr. 4.1



Obrázek 4.1: Mnou vytvořený design GUI

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

---

<sup>1</sup>Graphical User Interface

<sup>2</sup>atomic proměnná v c++ znamená, že k ní nemůžou dvě vlákna přistoupit zároveň

## 4.1 Návod na použití

Pro zkompileování 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 cmake
```

- Poté vytvoříte makefile pomocí cmake a zkompilujete projekt příkazem make.

```
cmake .  
make
```

- Nakonec spustíte program v adresáři src.

```
cd src  
./gui_chess
```

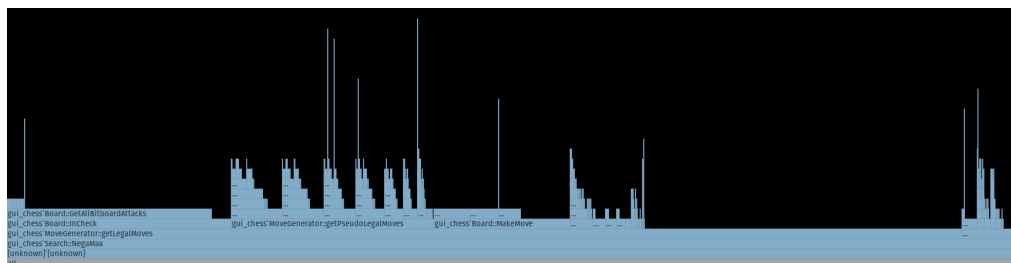
# Kapitola 5

## Optimalizování

Klíčový pro šachový engine je rychlost. Abychom měli šachový engine co nejrychlejší, musíme ho různými způsoby optimalizovat. Například použití rychlejšího algoritmu, efektivnější datové struktury či prosté odstranění redundantních operací.

Je důležité primárně optimalizovat funkce, které jsou nejvíce časově náročné, abychom z našeho úsilí vytěžili co nejvíce. Proto využijeme tzv. profiler. Jedná se o program, který analyzuje chod programu a umožní mi zjistit, kolik jaké funkce zabírají času. Použijeme na to linuxový program `perf`<sup>1</sup> s vizualizací 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 lze vidět, že nejvíc času zabírá 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. Šachový engine zvládne pomocí `perft` (viz kapitola 3.2.1) prohledat 4 865 609 pozic za přibližně jednu sekundu. Pro porovnání, Stockfish, nejlepší volně dostupný šachový engine na světě, prohledal stejný počet pozic za 58 milisekund, takže je ještě spousta prostoru, kam lze tento engine zlepšovat.



Obrázek 5.1: Plamenový graf mého šachového enginu.

<sup>1</sup>Program na analýzu běhu programu. Obsažený v linuxovém kernelu.

# Kapitola 6

## Závěr

Cílem této maturitní práce bylo vytvořit funkční šachový engine, popsat jeho části a celkově uvést do problematiky šachových enginů. Jelikož šachový engine, je už trošku větší projekt a v C++ jsem dělal pouze menší projekty, musel jsem poměrně velkou část práce na maturitní práci strávit rešerší a teoretickým návrhem abych nemusel projekt poté přepisovat.

Hlavním zdrojem vědomostí o šachových enginech byla stránka [chessprogramming.org](http://chessprogramming.org) a diplomová práce *Some aspects of chess programming*. [15] S herními boty už jsem měl zkušenost, protože jsem v druhém ročníku vytvořil bota na piškvorky.

Mezi největší problémy se kterými jsem se v této maturitní práci setkal, bylo že jsem dostával segmentation fault pouze ve verzi programu s optimalizacemi. To znamená, že jsem nemohl debugovat C++ instrukce, abych zjistil kde se stala chyba. Takže na vyřešení tohoto problému jsem musel analyzovat přímo assembly instrukce. Poté mi ještě docela zavařilo, když jsem rozmístil šachové figurky zrcadlově. Perft výsledky byli stejné, ale šachový engine potom hrál nesmyslné pohyby z databáze. Oba problémy jsem nakonec zdárně vyřešil.

Při některých designových rozhodnutích jsem udělal chybu, že jsem upřednostnil simplicitu před výkonem a nakonec to mělo větší následky než jsem očekával. Tento engine tedy není příliš konkurenceschopný proti ostatním šachovým enginům. Nedostatečný výkon enginu doháním poměrně rozsáhlou databází pohybů. Zparsoval jsem 1.9 GB šachových her do databáze pohybů o velikosti cca 21 MB.

Pro používání enginu jsem vytvořil velmi zjednodušené GUI, ve kterém jde hrát pouze za bílého a pro hraní znovu musíte GUI restartovat. GUI bylo používáno hlavně pro snadnější debugování.

Jako celek tento projekt hodnotím jako úspěch, ale pravděpodobně se k němu ještě

někdy vrátím a většinu pozměním. Zaprvé kvůli většímu výkonu, a zadruhé abych vyzkoušel různé experimentální techniky. Podle mého názoru je šachový engine ideální jako projekt na experimentování, protože se každou chvílí v oblasti šachových enginů stane nějaký průlom.

# Literatura

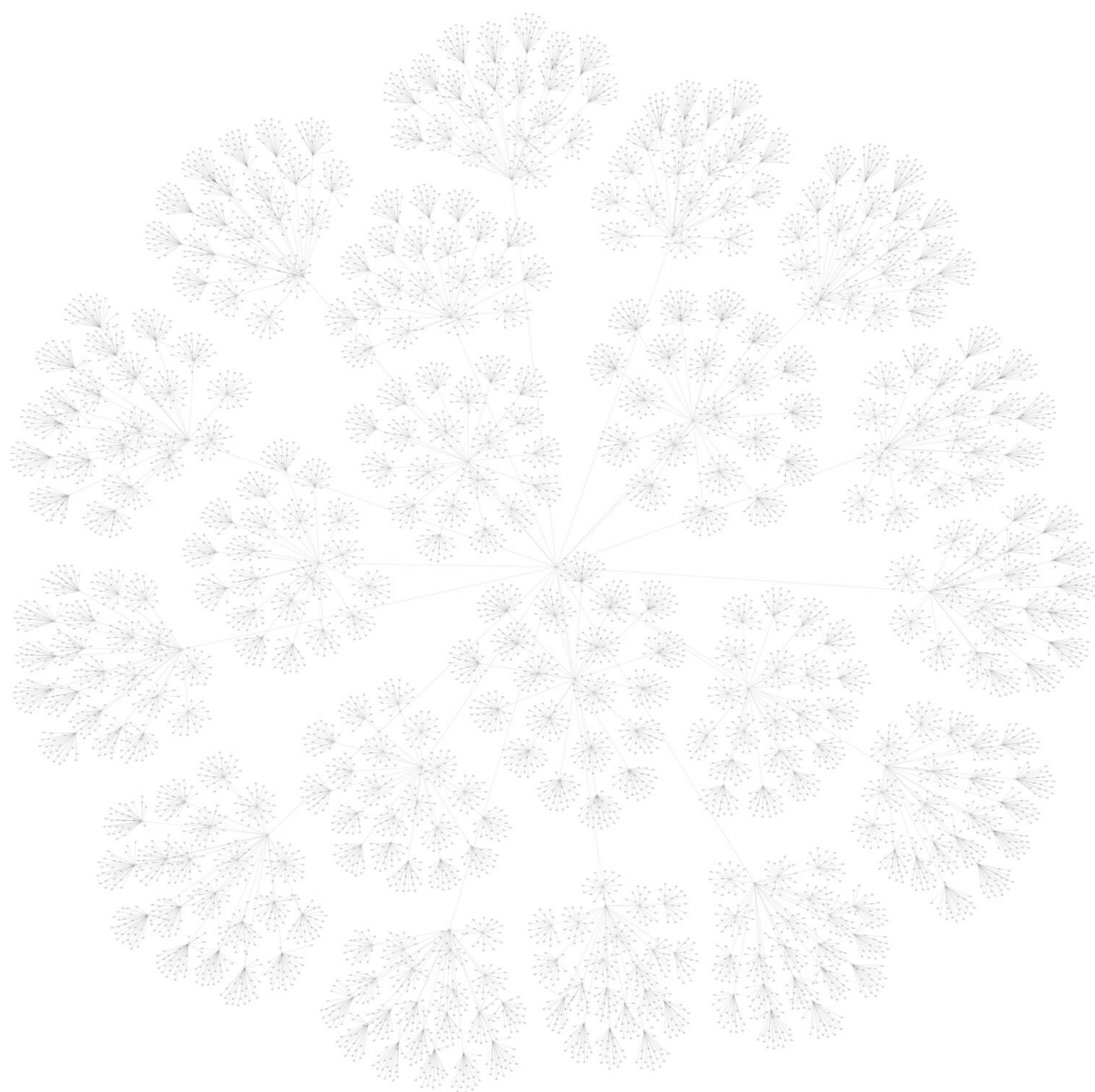
- [1] Claude, E. S.: Programming a Computer for Playing Chess. *Philosophical Magazine, Ser*, ročník 7, č. 41, 1950: str. 314.
- [2] David Barr: olc::PixelGameEngine. Dostupné na: [OneLoneCoder.com](http://OneLoneCoder.com).
- [3] Edwards, S. J.: Portable game notation specification and implementation guide. *Retrieved April*, ročník 4, 1994: str. 2011.
- [4] Gregg, B.: Dostupné na: <http://www.brendangregg.com/flamegraphs.html>, 2021.
- [5] Příspěvatelé Chess Programming Wiki: Alpha-Beta. Dostupné na: <https://www.chessprogramming.org/Alpha-Beta>, 2021.
- [6] Příspěvatelé Chess Programming Wiki: Bitboards. Dostupné na: <https://www.chessprogramming.org/Bitboards>, 2021.
- [7] Příspěvatelé Chess Programming Wiki: Move Ordering. Dostupné na: [https://www.chessprogramming.org/Move\\_Ordering](https://www.chessprogramming.org/Move_Ordering), 2021.
- [8] Příspěvatelé Chess Programming Wiki: Perft. Dostupné na: <https://www.chessprogramming.org/Perft>, 2021.
- [9] Příspěvatelé Chess Programming Wiki: Perft results. Dostupné na: [https://www.chessprogramming.org/Perft\\_Results](https://www.chessprogramming.org/Perft_Results), 2021.
- [10] Příspěvatelé Wikipedie: Game tree. Dostupné na: [https://en.wikipedia.org/w/index.php?title=Game\\_tree&oldid=1003093164](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.: Perf results. Dostupné na: <http://rebel13.nl/download/data.html>, 2021.
- [15] Streczek, M.: *Some aspects of chess programming*. Diplomová práce, Technical University of Łódź, 2004.
- [16] Yang, D.: Introducing NNUE Evaluation. Dostupné na <https://stockfishchess.org/blog/2020/introducing-nnue-evaluation/>.
- [17] 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 pohybech