



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

programátor

# 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 also 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

<b>1</b>	<b>Úvod</b>	<b>4</b>
<b>2</b>	<b>Popis a struktura šachového enginu</b>	<b>5</b>
2.1	Teorie her . . . . .	5
2.2	Dělení šachových enginů . . . . .	5
2.3	Struktura šachového enginu . . . . .	7
<b>3</b>	<b>Implementace</b>	<b>9</b>
3.1	Bitboardy . . . . .	9
3.2	Generování pohybů . . . . .	10
3.2.1	Debugování . . . . .	13
3.3	Prohledávání pozic . . . . .	14
3.3.1	Minimax . . . . .	15
3.3.2	Alpha-Beta pruning . . . . .	16
3.4	Transpoziční tabulka . . . . .	17
3.5	Databáze otevíracích pohybů . . . . .	18
3.5.1	PGN formát . . . . .	18
3.6	GUI . . . . .	20
<b>4</b>	<b>Závěr</b>	<b>21</b>
	<b>Přílohy</b>	<b>23</b>
<b>A</b>	<b>Příloha</b>	<b>24</b>

# Kapitola 1

## Úvod

Tato maturitní práce představuje můj šachový engine a popisuje různé techniky a algoritmy používané v šachových enginech.

V této práci nebudu popisovat pravidla šachů, budu počítat s vaší znalostí pravidel.

# 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 jsem vygeneroval graf pomocí programu Graphviz, který ukazuje jak se hra po třech tazích větví. Viz obr. 2.1

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  $35^{80} \approx 10^{123}$  [2]. 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. [10] Při perfektním zahrání od obou hráčů skončí hra remízou.

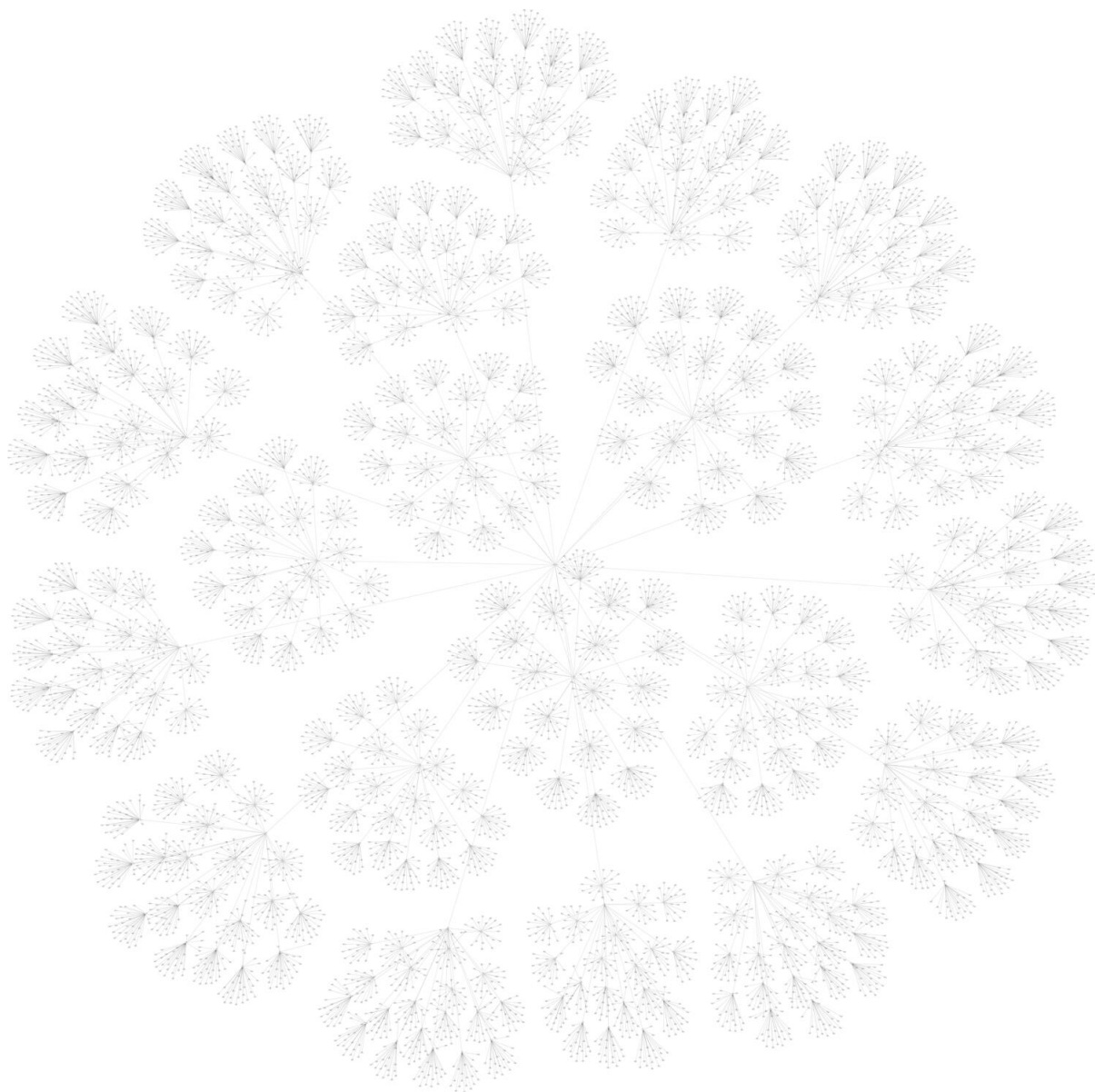
Šachy jsou:

- 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.

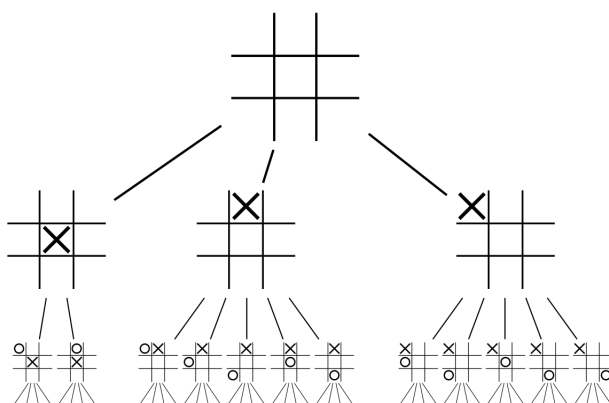
Díky tomu můžeme tvořit herní strom všech možných tahů do určité hloubky a relativně dobře odhadovat hodnoty pohybů. Herní strom vypadá přibližně takto. Viz obr. 2.2

### 2.2 Dělení šachových enginů

V dnešní době existují v podstatě dva druhy šachových botů.



Obrázek 2.1: větvení šachů po třech pohybech



Obrázek 2.2: herní strom piškvorek [8]

- 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 dohnány prohledáváním spousty pozic do velké hloubky.
  - Detaily budou v kapitole 3.3
- Šachový bot založený na neuronových sítích 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<sup>1</sup>, 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<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 ??
- Evaluace pozic
  - Hodnotí pozici podle hodnoty, pozice a struktury figurek.
  - Klíčová pro chování šachového enginu.
  - Na základě ní se rozhoduje v prohledávání pozic.
- Prohledávací

---

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

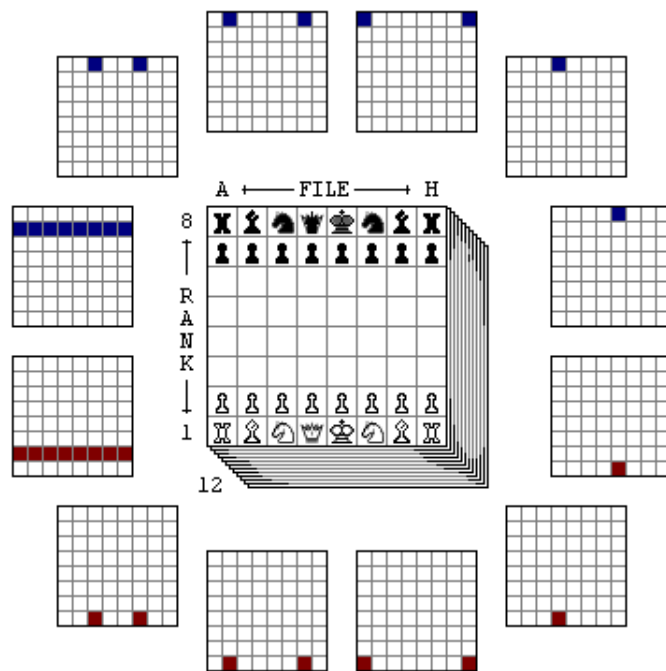
<sup>2</sup>Bitboard nebo bitmapa je 64 bitové číslo kde v jeho binární podobě každá jednička znamená zabranou pozici na šachovém poli.

# 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[4]

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

---

```

1  typedef unsigned long long bitboard;
2  class Board {
3      public:
4          bitboard all_bitboards[2][6]{};
5
6          // spojené pozice
7
8          // vraci bitboard figurek určité barvy
9          bitboard PiecesOfColor(bool color);
10
11         // vraci bitboard všech figurek
12         bitboard AllPieces();
13     }

```

---

## 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

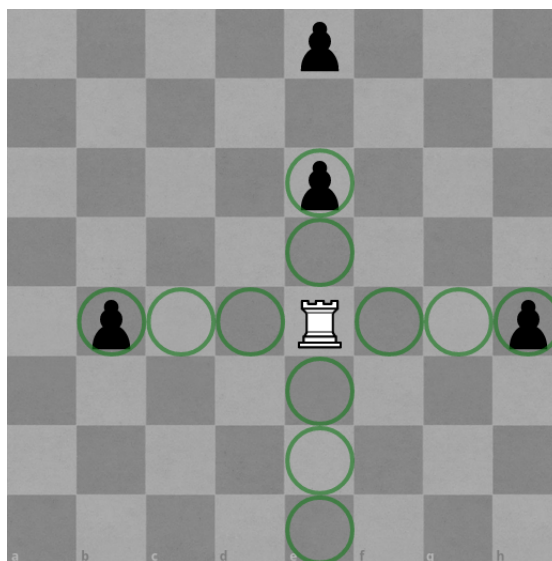
pro všech 64 pozic kde se může figurka nacházet si předpočítáme možné pozice kam může jít. Všechny tyto pozice uložíme do pole a jako index použijeme pozici bitu. Když potom narazíme na figurku na určitém poli, stačí vzít předpočítaný bitboard. Generování potom probíhá postupným procházením předpočítaného 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 přístě dostal další jedničku.

Na ukázkou 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 nacteného kone
11       my_knights ^= 1ULL << (from);
12       // utoky ze soucasneho kone
13       bitboard attacks = precomp.precomputed_knights[from];
14       // odstraneni pohybu, ktere by skončili 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  $2^{10} = 1024$  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 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. Já jsem si svoje čísla sám nepočítal.

Pouze jsem vzal čísla, které už někdo spočítal[9]

Ukážu na příkladu 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 jenom posledních 10 bitů čísla a máme index.

### 3.2.1 Debugování

Problém s debugováním 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)[6]. Perft projíždí všechny možné pozice do určité hloubky a konečné uzly stromu sečte. Potom výsledek porovná se známými správnými výsledky.

Výsledek spočítáme klasickým DFS (Depth First Search). Budeme rekurzivně volat naši funkci na všechny 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 moji 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         return gen.getLegalMoves(board).size();
8     }
9     unsigned long long nodes = 0;
10    for (auto move : gen.getLegalMoves(board)) {
11        Board next_board = board;
12        next_board.MakeMove(move);
13        nodes += perft(depth - 1, next_board);
14    }
15
16    return nodes;
17 };
```

---

Tuto funkci potom využiji na ověření správnosti generátoru takto. Správné hodnoty konečných pozic jsem sebral z internetu.[7] Poté jsem jenom pomocí c++ knihovny Catch2 porovnal hodnoty.

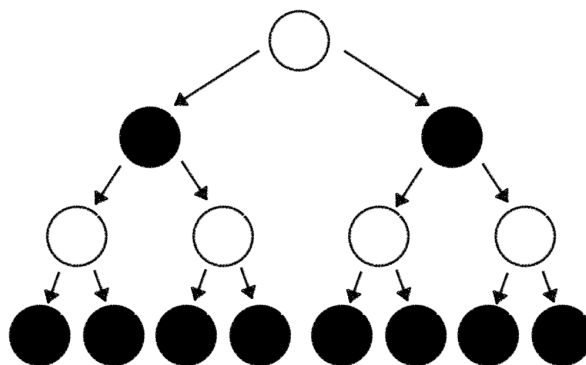
---

```
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);
```

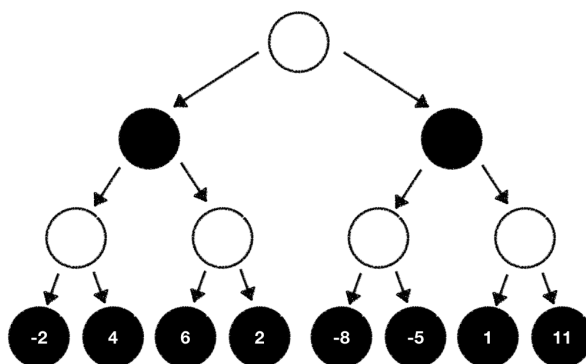
---

### 3.3 Prohledávání pozic

Alfa beta pruning je metoda na zvolení nejlepšího pohybu ve stromu pohybů a odřezávání nepotřebných větví. Je to vylepšení algoritmu minimax.



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



Obrázek 3.4: Minimax strom. Poslední pozice ohodnocené.

### 3.3.1 Minimax

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





## 3.4 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`[12]. 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 zahraničního 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 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. Zadrugé 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.

## 3.5 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[11], což je databáze 2.5 milionu kvalitních šachových partií ve formátu PGN[3].

### 3.5.1 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 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 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 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

---

Svůj PGN parser<sup>1</sup> chci mít co nejjednodušší, takže budu ignorovat informační tagy<sup>2</sup> 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 výsledky ukladat do souborů. Nejdříve si načtu 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. 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 řádku všechny pohyby které byli v dané pozici 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 uloženy jako string ve proměnné `tok`. Nejdřív zvětším hloubku prohledávání o jednu. Potom přidám jedničku ke četnosti pohybu, který se chystám zahrát v dalším příkazu.

---

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

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

# Kapitola 4

## GUI

Pro vytvoření GUI<sup>1</sup> mého programu jsem zvolil `OLC::PixelGameEngine[?]`, 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. 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. Mám jednu `atomic`<sup>2</sup> proměnnou, která rozhoduje v jaké fázi je program, tedy jestli je na řadě bot, či hráč.

### 4.1 Závěr

---

<sup>1</sup>Graphical User Interface

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

# Seznam tabulek

# Seznam obrázků

2.1	větvění šachů po třech pohybech . . . . .	6
2.2	herní strom piškvorek [8] . . . . .	7
3.1	Reprezentace herní plochy pomocí bitboardů[4] . . . . .	10
3.2	Možnosti pohybu věže . . . . .	12
3.3	Minimax strom. Barva uzlu určuje hráče na tahu v dané pozici . . . . .	15
3.4	Minimax strom. Poslední pozice ohodnocené. . . . .	15
3.5	Minimax strom. Všechny pozice ohodnocené . . . . .	16
3.6	Ohodnocený strom pomocí alfa-beta pruningu. Prázdné uzly byli ořezány, jelikož je nebylo třeba prohledávat . . . . .	16

Příloha A

Příloha



# Literatura

- [1] Apache Software Foundation: Hadoop.  
URL <https://hadoop.apache.org>
- [2] Claude, E. S.: Programming a Computer for Playing Chess. *Philosophical Magazine, Ser*, ročník 7, č. 41, 1950: str. 314.
- [3] Edwards, S. J.: Portable game notation specification and implementation guide. *Retrieved April*, ročník 4, 1994: str. 2011.
- [4] Příspěvatelé Chess Programming Wiki: Bitboards. Dostupné na: <https://www.chessprogramming.org/Bitboards>, 2021.
- [5] Příspěvatelé Chess Programming Wiki: cMove Ordering. Dostupné na: [https://www.chessprogramming.org/Move\\_Ordering](https://www.chessprogramming.org/Move_Ordering), 2021.
- [6] Příspěvatelé Chess Programming Wiki: Perft. Dostupné na: <https://www.chessprogramming.org/Perft>, 2021.
- [7] Příspěvatelé Chess Programming Wiki: Perft results. Dostupné na: [https://www.chessprogramming.org/Perft\\_Results](https://www.chessprogramming.org/Perft_Results), 2021.
- [8] 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.
- [9] Rustad-Elliott, R.: Fast Chess Move Generation With Magic Bitboards. Dostupné na <https://rhysre.net/fast-chess-move-generation-with-magic-bitboards.html>.
- [10] Schaeffer, J.; Björnsson, Y.; Kishimoto, A.; aj.: Checkers Is Solved. *Science*, ročník 317, 10 2007: s. 1518–1522.
- [11] Schroder, E.: Perft results. Dostupné na: <http://rebel13.nl/download/data.html>, 2021.

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