
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

Děkuju pani Maříkový.

# Obsah

# Kapitola 1

# Úvod

Tato maturitní práce představuje můj šachový engine a popisuje různé techniky, či 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 stálé nevyřešená hra tzn. neví se jak vypadá bezchybná hra. Jsou nevyřešené, protože počet možných pozic je enormní. Už jenom po třech pohybech je 8902 možných variant hry. Pro představu jsem vygeneroval graf pomocí programu Graphviz, který ukazuje jak se hra po třech tazích větví. Viz obr. 2.1

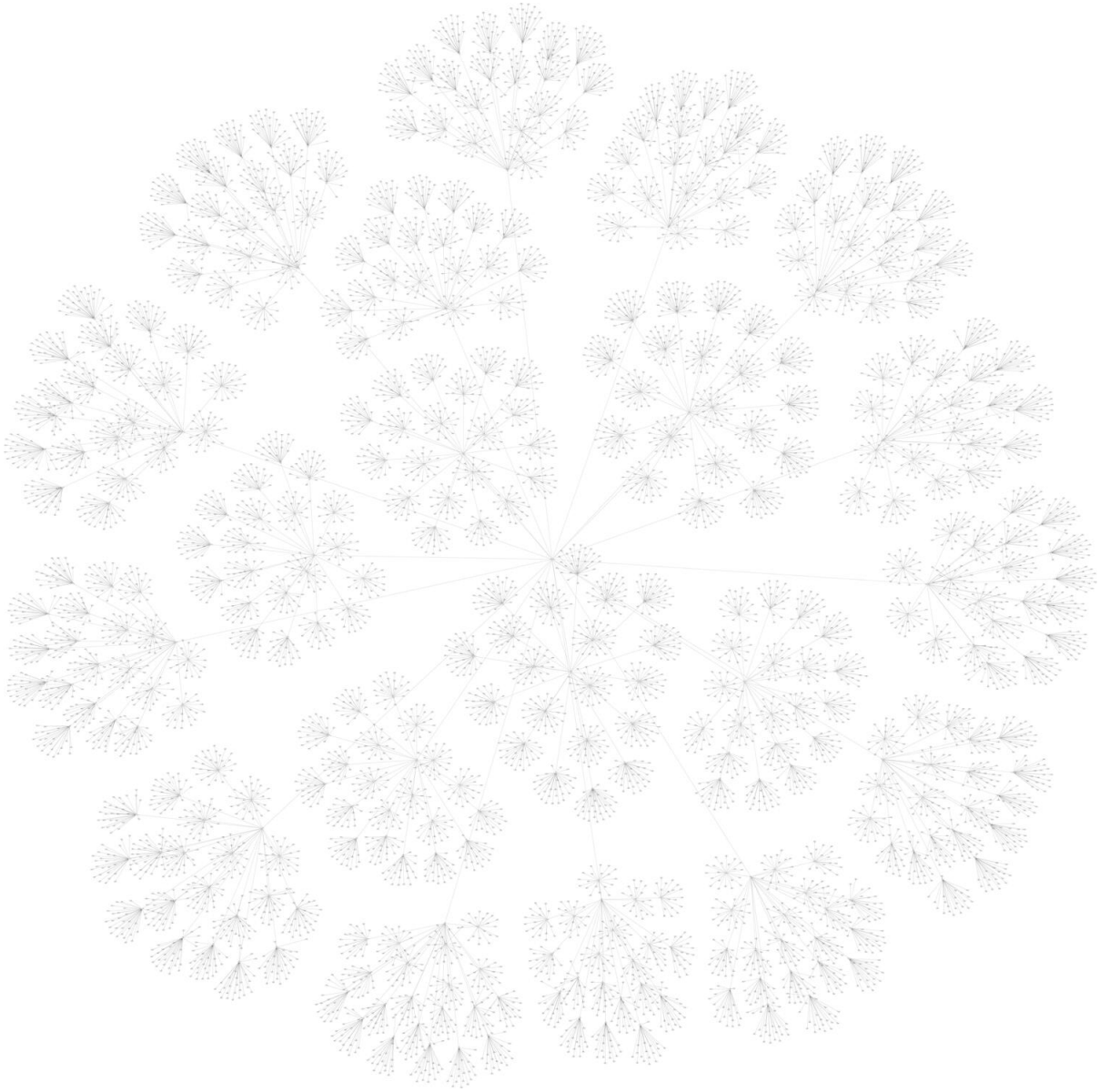
Průměrný počet pohybů ve všech situací je přibližně 35, průměrná hra má 80 pohybů, takže když chceme dostat hodně nepřesný odhad možných šachovým partií, tak nám stačí tyto dvě hodnoty umocnit $35^{80} \approx 10^{123}$[1]. Z tohoto enormního čísla vyplývá, že šachy nelze vyřešit hrubou silou a pravděpodobně šachy v nejbližší době, či dokonce nikdy nevyřešíme. Pro zajímavost dáma má přibližně $5 * 10^{20}$ variací her a byla vyřešena v roce 2007.[7] 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.

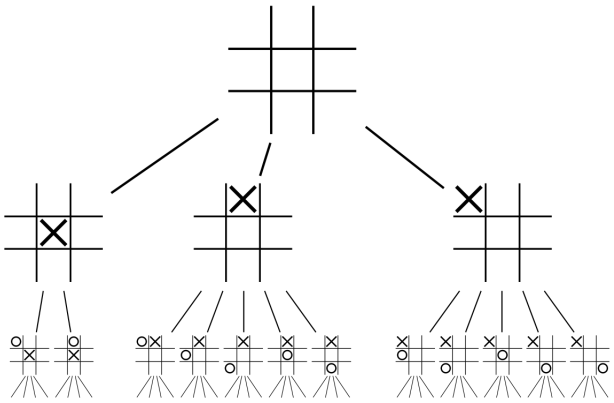
Díky tomu můžeme tvořit herní strom všech možných tahů do určité hloubky a relativně dobře odhadovat hodnoty pohybů. Herní strom vypadá přibližně takto. Viz obr. 2.2

## 2.2 Dělení šachových enginů

V dnešní době existují v podstatě dva druhy šachových botů.



Obrázek 2.1: větvění šachů po třech pohyběch



Obrázek 2.2: herní strom piškvorek [5]

- 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 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.
  - Na základě ní se rozhoduje v prohledávání pozic.
- Prohledávací

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

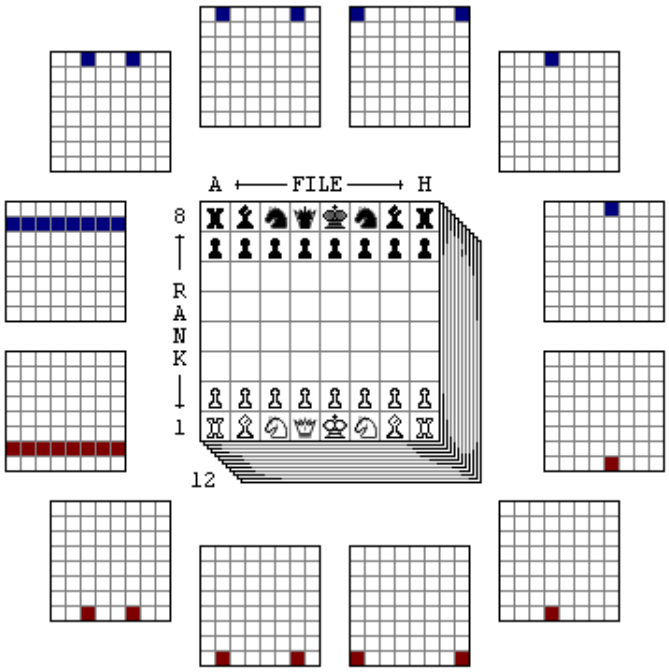
[2]Bitboard nebo bitmapa je 64 bitové číslo kde v jeho binární podobně každá jednička znamená zabranou pozici na šachovém poli.

# 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[2]

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();
}
```

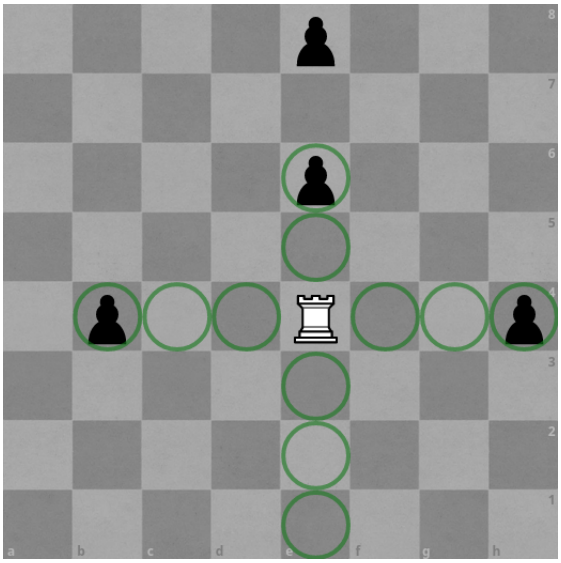
## 3.2 Generování pohybů

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 $2^{10} = 1024$ 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 prvnich x bitů. V minulém případě by to posunulo těch 10 bitů na začátek čísla. Pak jenom oříznememe 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[6]

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)[3]. 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.

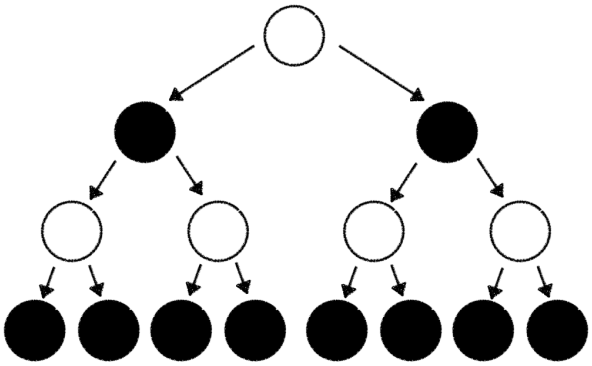
```
unsigned long long perft(int depth, Board &board) {
    MoveGenerator gen = MoveGenerator();
    if (depth == 0) {
        return 1;
    }
    if (depth == 1) {
        return gen.getLegalMoves(board).size();
    }
    unsigned long long nodes = 0;
    for (auto move : gen.getLegalMoves(board)) {
        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.[4] Poté jsem jenom pomocí c++ knihovny Catch2 porovnal hodnoty.

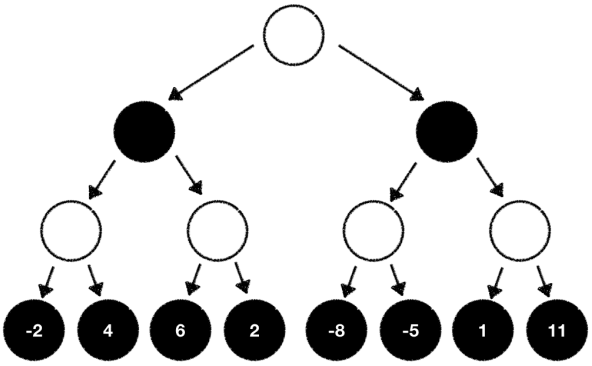
```
    REQUIRE(perft(1, board) == 20);
    REQUIRE(perft(2, board) == 400);
    REQUIRE(perft(3, board) == 8902);
    REQUIRE(perft(4, board) == 197281);
    REQUIRE(perft(5, board) == 4865609);
    REQUIRE(perft(6, board) == 119060324);
```

## 3.3 Alpha–beta pruning

Alfa beta prunning 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. Ohodnocené

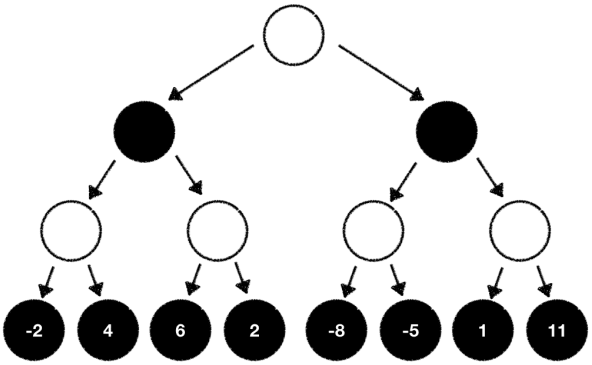
### 3.3.1 Minimax

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




Obrázek 3.5: Minimax strom. Ohodnocené

# Kapitola 4

# Závěr

# Seznam tabulek

# Seznam obrázků

# Příloha A

# Příloha

# Literatura


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

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

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

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

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

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

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