Katedra informatiky
Přírodovědecká fakulta
Univerzita Palackého v Olomouci

DIPLOMOVÁ PRÁCE

Implementace a porovnání pokročilých metod konstrukce šachového enginu

2021

Bc. Lukáš Vyhnálek

Vedoucí práce:
Mgr. Petr Osička, Ph.D.

Studijní obor: Aplikovaná informatika,
prezenční forma

Bibliografické údaje

Autor:	Bc. Lukáš Vyhnálek
Název práce:	Implementace a porovnání pokročilých metod konstrukce šachového enginu
Typ práce:	diplomová práce
Pracoviště:	Katedra informatiky, Přírodovědecká fakulta, Univerzita Palackého v Olomouci
Rok obhajoby:	2021
Studijní obor:	Aplikovaná informatika, prezenční forma
Vedoucí práce:	Mgr. Petr Osička, Ph.D.
Počet stran:	75
Přílohy:	1 CD/DVD
Jazyk práce:	český

Bibliograhic info

Author:	Bc. Lukáš Vyhnálek
Title:	Implementation and comparison of advanced methods of chess engine construction
Thesis type:	master thesis
Department:	Department of Computer Science, Faculty of Science, Palacký University Olomouc
Year of defense:	2021
Study field:	Applied Computer Science, full-time form
Supervisor:	Mgr. Petr Osička, Ph.D.
Page count:	75
Supplements:	1 CD/DVD
Thesis language:	Czech

Anotace

V práci jsem se zabýval zlepšením počítačového hráče šachu, který byl vytvořen jako bakalářská práce. Nastudoval jsem si architekturu a jednotlivé heuristiky existujících výkonných enginů. Některé z heuristik jsem následně implementoval a otestoval jejich vliv na výkon šachového enginu.

Synopsis

In this thesis I improved the chess engine that was created as a bachelor's thesis. I studied the architecture and heuristics used in powerful chess engines. Furthermore I implemented some of the heuristics and I have tested their impact on the performance of the engine.

Klíčová slova: bitové šachovnice, engine, en passant, figura, hloubka

Keywords: bitboards, engine, en passant, chess piece, depth

Děkuji panu Mgr. Petru Osičkovi Ph.D. za vedení této diplomové práce

Místopřísežně prohlašuji, že jsem celou práci včetně příloh vypracoval/a samostatně a za použití pouze zdrojů citovaných v textu práce a uvedených v seznamu literatury.

datum odevzdání práce podpis autora

Obsah

Seznam obrázků

Seznam zdrojových kódů

1 Úvod

1.1 Náplň práce

V této diplomové práci jsem se zabýval využitím pokročilých metod konstrukce šachového enginu pro zlepšení výkonu enginu. Při implementaci jsem jednotlivé funkce otestoval, aby si čtenář mohl udělat obrázek o vlivu dané funkcionality na výkon enginu.

1.2 Použití Enginů

Šachový engine je program, který pro danou pozici spočítá ideální tah. U výkonných enginů je dodaný tah často nejlepší možný. Proto se v dnešní době používají šachové enginy například k rozboru partií kdy se snaží odhalit chyby, které hráč udělal. Z pohledu historie je tvorba šachových enginů bohatá. Svůj šachový engine vytvořili například i Alan Turing nebo Claude Shannon. Zpočátku se ovšem enginy nedokázaly člověku vyrovnat. Trvalo to přibližně 50 let, než se první enginy mohly měřit s profesionálními hráči šachu. V dnešní době už ale člověk nemá šanci. Šachové enginy jako Stockfish nebo AlphaZero jasně dominují. Šachový engine Stockfish má odhadované elo okolo 3692 bodů, zatímco mistr světa Magnus Carlsen má elo pouze 2862 bodů.

1.3 Základ Enginu

Diplomová práce je pokračováním mé bakalářské práce, kde jsem implementoval šachový engine, který měl elo okolo 1700 bodů. Při tvorbě diplomové práce jsem stavěl již na funkčním enginu a využíval části a funkcionality, které jsou již popsány v textu bakalářské práce.

Engine využívá minimax vyhledávání s alphabeta ořezáváním, transpoziční tabulku a late move reduction heuristiku.

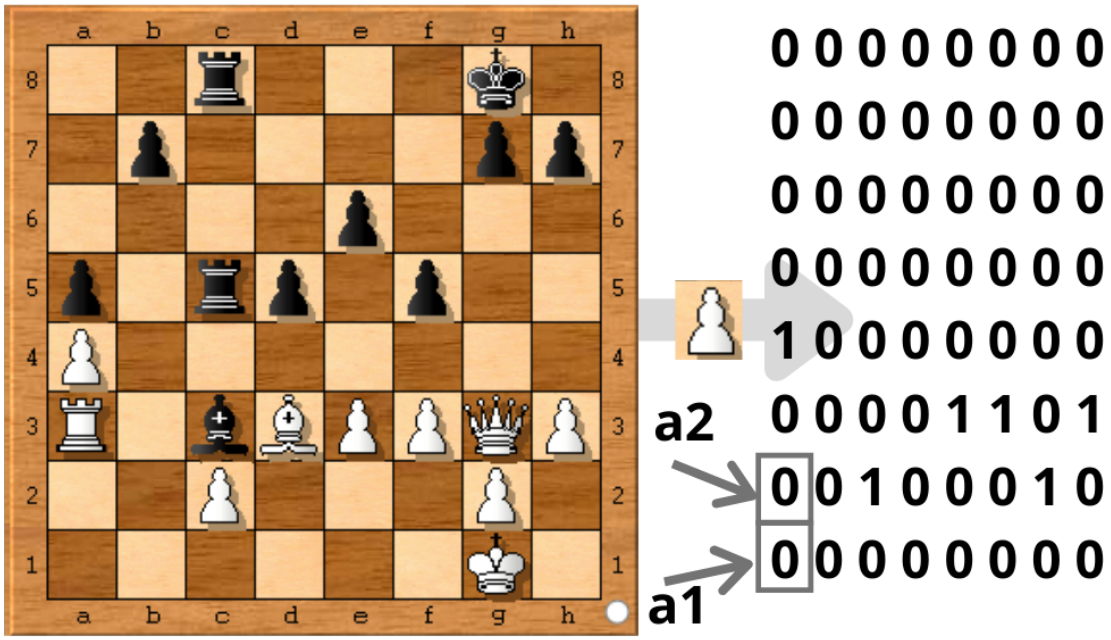
Function Name	Total CPU [unit,...	Self CPU [unit, %]	Module
Quiesce	6691 (59.78%)	136 (1.22%)	ChessEngine.exe
rateBoard	4289 (38.32%)	54 (0.48%)	ChessEngine.exe
beginningGameRating	4138 (36.97%)	186 (1.66%)	ChessEngine.exe
bonusPossitionRate	1846 (16.49%)	420 (3.75%)	ChessEngine.exe
initializeTable	1832 (16.37%)	424 (3.79%)	ChessEngine.exe
ChessEngine::ChessEngine	1832 (16.37%)	0 (0.00%)	ChessEngine.exe
to64index	1656 (14.79%)	1524 (13.62%)	ChessEngine.exe
mobilityRate	1287 (11.50%)	1227 (10.96%)	ChessEngine.exe
Board::GenerateCapturingMoves	975 (8.71%)	16 (0.14%)	ChessEngine.exe
Board::GenerateMoves	935 (8.35%)	7 (0.06%)	ChessEngine.exe
simpleHash	899 (8.03%)	261 (2.33%)	ChessEngine.exe
Move::Move	872 (7.79%)	872 (7.79%)	ChessEngine.exe
Transposition::Transposition	809 (7.23%)	375 (3.35%)	ChessEngine.exe
GenerateCastelingMoves	625 (5.58%)	6 (0.05%)	ChessEngine.exe
OpponentThreathens	609 (5.44%)	164 (1.47%)	ChessEngine.exe
pawnRatingWithKingSafety	559 (4.99%)	482 (4.31%)	ChessEngine.exe
Board::PlayMove	508 (4.54%)	75 (0.67%)	ChessEngine.exe
MoveList::MoveList	455 (4.07%)	23 (0.21%)	ChessEngine.exe
Board::~Board	358 (3.20%)	148 (1.32%)	ChessEngine.exe
__CheckForDebuggerJustMyCode	355 (3.17%)	355 (3.17%)	ChessEngine.exe
mirrorIndex	334 (2.98%)	306 (2.73%)	ChessEngine.exe
GeneratePawnCaptureMoves	316 (2.82%)	105 (0.94%)	ChessEngine.exe
isCheck	273 (2.44%)	267 (2.39%)	ChessEngine.exe
MoveList::~MoveList	260 (2.32%)	222 (1.98%)	ChessEngine.exe
Board::UndoMove	256 (2.29%)	64 (0.57%)	ChessEngine.exe
PickBestMove	252 (2.25%)	155 (1.38%)	ChessEngine.exe

Obrázek 1: Výsledek performance profiler analýzy

2 Bitové šachovnice

Jedna z nejvíce vytěžovaných částí kódu je kód třídy Board. Tato třída se stará o generování a hraní tahů. Další část kódu, kde engine stráví velké množství času, je ohodnocující funkce. Tato funkce pracuje s instancí třídy Board a zjišťuje, která strana má lepší postavení.

Již z tohoto vyplývá, že způsob, jakým šachovnici reprezentujeme v programu, má velký vliv na výkon enginu. V bakalářské práci jsem zvolil reprezentaci pomocí pole. Především proto, že je jednodušší na implementaci. Na konci bakalářské práce jsem se zmínil o bitových šachovnicích, které by mohly engine značně zlepšit. Jak si může čtenář všimnout na obrázku 1, generování tahů a ohodnocení pozice zabere enginu většinu času na výpočet. Metoda `GenerateCapturingMoves` zabírá 8.71%. Medota `GenerateMoves` 8.35% a ohodnocení pozice 38.32%.



Obrázek 2: Bitová šachovnice reprezentující bílé pěšce

2.1 Co jsou to bitové šachovnice?

Bitové šachovnice představují způsob, jakým můžeme v počítači reprezentovat šachovnici. Na šachovnici se nachází 64 polí. To znamená, že můžeme využívat sérii 64 bitových čísel pro reprezentaci šachovnice tak, že každý bit reprezentuje právě jedno pole. Například nejméně významný bit koresponduje s polem a1. Druhý nejméně významný bit s polem b1 atd. Šachovnice je v šedesátičtyřbitovém čísle umístěna po řádcích, kde každý řádek představuje 8 bitů. Například nejnižších osm bitů reprezentuje řádek číslo jedna. Pokud se figura na poli nachází, je korespondující bit roven jedné. Jinak je roven nule.

U bitových šachovnic provádíme několik základních operací:

- Nastavení bitu na hodnotu 1.
- Operaci COUNT, která vrátí počet bitů rovných 1.
- Operaci POP, která nastaví nejméně významný bit, který má hodnotu jedna na nulu a vrátí pozici tohoto bitu jako výsledek.

Nastavení bitu na hodnotu jedna se používá například při tahu figurou, kdy potřebujeme uložit novou pozici figury. Nastavení bitu se dá jednoduše provést pomocí operace OR.

Operace COUNT se používá například při evaluaci. Operace bere 64 bitové číslo jako argument a vrací počet bitů nastavených na 1. Operace by se dala

implementovat pomocí brute force algoritmu, jinými slovy by se iterativně procházelo přes všechny bity. Nicméně existuje i rychlejší způsob, kdy děláme operaci AND mezi bitovou šachovnicí a dekrementovanou bitovou šachovnicí. Výsledkem této operace je původní bitová šachovnice, která má na nejméně významném bitu, na kterém původně byla jednička, nulu.

Operaci POP používáme, když iterujeme přes bitovou šachovnici. Například při generování tahů, když tahy generujeme pro jednotlivé pěšce zvlášť, ale všechny pěšce reprezentujeme jednou bitovou šachovnicí. Operace POP bere jako argument odkaz na bitovou šachovnici. Nastaví nejméně významný bit, který má hodnotu jedna na nulu a vrátí pozici tohoto bitu jako výsledek.

```
1  char POP(U64 *bb) {
2   U64 b = *bb ^ (*bb - 1);
3   unsigned int fold = (unsigned)((b & 0xffffffff) ^ (b >> 32));
4   *bb &= (*bb - 1);
5   return BitTable[(fold * 0x783a9b23) >> 26];
6  }
7
8  char COUNT(U64 b) {
9   int r;
10  for (r = 0; b; r++, b &= b - 1);
11  return r;
12 }
```

Zdrojový kód 1: Operace POP a COUNT

2.2 Reprezentace

Uchovat celou šachovnici v jednom 64bitovém čísle není možné. Je potřeba mít bitovou šachovnici pro každý typ figury, který se na šachovnici může vyskytovat. Máme tedy jednu bitovou šachovnici pro bílého krále, další bitovou šachovnici pro bílé dámy atd. Pro reprezentaci všech figur potřebujeme celkem 12 bitových šachovnic. Ty jsou uloženy ve dvou polích, první obsahuje bitové šachovnice reprezentující figury bílého hráče `whitePiecesList`. Druhé obsahuje bitové šachovnice reprezentující figury černého hráče `blackPiecesList`. Dále máme dvě pomocné bitové šachovnice, ve kterých uchováváme informace o pozicích všech bílých figur (`whitePieces`) a všech černých figur (`blackPieces`) pro rychlejší generování tahů. V neposlední řadě potřebujeme jednu bitovou šachovnici (`enPassant`) na reprezentaci pole pro braní mimochodem.

```
1   unsigned long long whitePieces = 0
       b0000000000000000000000000000000000000000000000001111111111111111
       ;
2   unsigned long long blackPieces = 0
       b1111111111111111000000000000000000000000000000000000000000000000
       ;
3
4   bool whiteOnMove = false;
5   char whiteCastleAlowed = BothSides;
6   char blackCastleAlowed = BothSides;
7   unsigned long long enPassant = 0;
8
9   unsigned long long tHelpAllTime = 0;
10  unsigned int tCollisionAllTime = 0;
11
12  char numberOfMovesWithoutTake = 0;
13  MoveList history = MoveList(MAX_HISTORY_MOVES);
14  Move bestMove;
15  unsigned long long blackPiecesList[6];
16  unsigned long long whitePiecesList[6];
```

Zdrojový kód 2: Reprezentace bitové šachovnice

2.3 Generování tahů

Je potřeba, aby generování tahů bylo rychlé, protože tahy generujeme v každé pozici, kterou procházíme. Ve funkcích, které se starají o generování tahů, jsem se tedy snažil využívat pouze binární operace.

Všeobecně generujeme tahy pomocí bitových posunů aplikovaných na jednotlivé bitové šachovnice. Generujeme tedy tahy pro všechny figury naráz. Neiterujeme přes jednotlivé figury. Engine má definované konstanty pro základní směry posunu figur. Tyto konstanty jsou následně hojně využívány při generování tahů. Například posun bílých pěšců o jeden řádek provedeme pomocí `(pawns << UP)`.

Po vygenerování možných pozic potřebujeme odstranit nevalidní tahy. Bitový posun totiž nebere v potaz pozici ostatních figur. Když tedy posuneme například pěšce dopředu, musíme ještě odstranit pole, na kterých se již nachází jiná figura. Zde využijeme dříve definované dvě pomocné bitové šachovnice, které reprezentují pozici všech bílých figur a všech černých figur. Pro odstranění nevalidních tahů použijeme bitový součin a komplement těchto šachovic. Například přípustné tahy při posunu bílého pěšce vygenerujeme `(pawns << UP)& (~opponentPieces )& (~myPieces)`.

Dále je třeba rozpoznat tahy, které berou protivníkovi figuru. Opět využijeme bitové operace AND mezi bitovou šachovnicí reprezentující možné tahy a bitovou šachovnicí reprezentující protivníkovy figury.

2.3.1 Generování tahů pěšcem

Pěšec je co se týče generování tahů nejsložitější figura. Má mnoho speciálních charakteristik, které ostatní figury nemají. Při generování používáme operace, které jsme si definovali v předchozí kapitole. Funkce pro generování tahů bere následující argumenty:

- Odkaz na seznam tahů, do tohoto seznamu přidává možné tahy.
- Bitovou šachovnici obsahující informace o pozici pěšců pro, které generujeme tahy.
- Bitovou šachovnici obsahující informace o pozici figur hráče, který je na tahu.
- Bitovou šachovnici obsahující informace o pozici figur hráče, který není na tahu.
- Přepínač indikující, zda je na tahu bílý nebo černý.
- Bitovou šachovnici obsahující en passant pole.

Nejdříve vygenerujeme bitovou šachovnici `advance` obsahující pozice pešců po provedení posunu pěšců o jednu řadu. Pomocí `advance & ROW_8`, kde `ROW_8` je bitová šachovnice mající jedničky pouze na řádku 8, získáme bitovou šachovnici `transforms`, která obsahuje pozice pešců, kteří budou transformováni na jiné figury. Pomocí operace POP pak projdeme `transforms` a `advance & ~ROW_8`, vygenerujeme tahy a přidáme je do seznamu všech vygenerovaných tahů.

Následně vygenerujeme brací tahy pro levou a pravou stranu. Nejdříve vygenerujeme možné brací tahy pro levou stranu `left`. Při generování musíme odebrat pěšce, které se nachází na sloupci H. K tomu slouží bitová šachovnice `NOT_H_FILE`, která obsahuje jedničky všude krom sloupce H, kde se nachází nuly. Provedeme bitový posun představující posun pěšců doleva nahoru a následně vybereme pouze tahy, které berou protivníkovi figuru. Zde je třeba brát v potaz i pole `enPassant`, pokud se pěšcem posuneme na toto pole, provedeme braní mimochodem. Jedná se tedy o validní tah, i když se na poli nenachází protivníkova figura. Dále je třeba rozlišit, zda se při tahu dostaneme na osmou řadu. Opět k tomu používáme bitovou šachovnici `ROW_8`. Následně procházíme bitové šachovnice `transforms` a `left & ~ROW_8` a postupně přidáváme tahy do seznamu všech vygenerovaných tahů. Generování tahů pro pravou stranu provádíme obdobně.

V neposlední řadě musíme vygenerovat tahy posunu pěšcem o dvě řady. Pomocí `pawns & WHITE_PAWNS_INIT_POSSITION` vybereme pouze pěšce, kteří se nachází na druhé řadě. K tomu slouží bitová šachovnice `WHITE_PAWNS_INIT_POSSITION` , která obsahuje jedničky pouze na druhém řádku. Provedeme bitový posun o jeden řádek a zkontrolujeme, zda je pole prázdné. Pokud se na poli nachází figura, není možné posunout pěšce o dvě řady. Předchozí krok opakujeme ještě jednou a výsledkem je bitová šachovnice obsahující přípustné tahy pěšcem o dvě řady.

```
1     U64 advance = (pawns << UP) & (~opponentPieces) & (~myPieces);
2     U64 transforms = advance & ROW_8;
3     advance &= ~ROW_8;
4     Move mv;
5
6     while (advance) {
7       char to = POP(&advance);
8       mv.from = to - UP;
9       mv.to = to;
10      mv.enPassant = 0;
11      mv.taken = 0;
12      mv.transforms = EMPTY;
13      mv.movedPiece = whitePawn;
14      (*moves).pushMove(mv);
15    }
16    // on the 8th row so it transforms
17    while (transforms) {
18      char to = POP(&transforms);
19      for (int i = whiteKnight; i < whiteKing; i++) {
20        mv.from = to - UP;
21        mv.to = to;
22        mv.enPassant = 0;
23        mv.transforms = i;
24        mv.score = value[i] - PAWN_VAL;
25        mv.movedPiece = whitePawn;
26        (*moves).pushMove(mv);
27      }
28    }
```

Zdrojový kód 3: Generování posunů pěšcem

2.3.2 Generování tahů střelcem

Jedna z výhod bitových šachovnic je možnost generování možných tahů pro všechny figury najednou. Při jednoduché implementaci, například pomocí pole, se tahy generují pro každou figuru zvlášť. To samozřejmě zpomaluje celý engine.

Při generování tahů střelcem je algoritmus jednoduchý: pro každý možný směr (například doprava nahoru) posuň všechny figury. Odstraň nevalidní tahy. Zjisti tahy, které berou protivníkovu figuru. Následně všechny přípustné tahy přidej do pole validních tahů.

Například pro směr doleva nahoru si nejdříve vytvoříme bitovou šachovnici `temp`, která je nastavena na momentální pozici střelců. Následně v cyklu z bitové šachovnice `temp` odebereme střelce, kteří se nacházejí na sloupci H a na osmé řadě. K tomu nám slouží kontanty `NOT_H_FILE` a `ROW_8`, které jsme si definovali u generování tahů pěšcem. Následně aplikujeme posun doleva a ostraníme pole, na kterých se již nachází figura hráče na tahu. Pomocí `temp & opponentPieces` vygenerujeme brací tahy, které následně přidáváme pomocí operace POP do seznamu všech tahů. Pomocí `temp &= ~opponentPieces` upravíme proměnou temp

```
1     U64 left = ((pawns & NOT_H_FILE) << LEFT_UP) & (opponentPieces |
          enPassant);
2     transforms = left & ROW_8;
3
4     while (left) {
5       char to = POP(&left);
6       mv.from = to - LEFT_UP;
7       mv.to = to;
8       mv.enPassant = 0;
9       mv.transforms = EMPTY;
10      mv.score = 100;
11      mv.movedPiece = whitePawn;
12      (*moves).pushMove(mv);
13    }
14    while (transforms) {
15      char to = POP(&transforms);
16      for (int i = whiteKnight; i < whiteKing; i++) {
17        mv.from = to - LEFT_UP;
18        mv.to = to;
19        mv.enPassant = 0;
20        mv.transforms = i;
21        mv.score = value[i] + 100;
22        mv.movedPiece = whitePawn;
23        (*moves).pushMove(mv);
24      }
25    }
```

Zdrojový kód 4: Generování bracích tahů pěšcem

pro další iteraci a opět přidáme přípustné tahy do seznamu všech tahů. Proces opakujeme, dokud se na bitové šachovnici `temp` nachází alespoň jeden střelec.

Generování tahů střelcem pro ostatní směry je implementováno podobným způsobem.

2.3.3 Generování tahů jezdcem

Jak už tomu u šachových enginů bývá, generování tahů jezdcem je nejjednodušší. Figura může totiž "skákat" přes ostatní figury, vždy je tedy předem dané, kam se jezdec může dostat.

Při generování tahů jsem se rozhodl použít pole o 64 elementech, kdy každý element představuje bitovou šachovnici možných tahů pro dané pole. Například první element je bitová šachovnice pro pole a1. Druhý element je pro pole b1 a tak dále.

Generování tahů potom probíhá následovně: pomocí operace pop zjistíme pozici jednoho z jezdců na šachovnici. Tuto pozici použijeme jako index u výše definovaného pole. Následně nám stačí odstranit pozice, na kterých se nachází figura stejné barvy.

```
1     U64 advance2 = ((((pawns & WHITE_PAWNS_INIT_POSSITION) << UP) &
          (~opponentPieces) & (~myPieces)) << UP) & (~opponentPieces) &
           (~myPieces);
2     while (advance2) {
3       char to = POP(&advance2);
4       mv.from = to - UP * 2;
5       mv.to = to;
6       mv.enPassant = 1ULL << (to - UP);
7       mv.transforms = EMPTY;
8       mv.movedPiece = whitePawn;
9       (*moves).pushMove(mv);
10    }
```

Zdrojový kód 5: Generování posunů pěšcem o dvě řady

Opět pro pozdější řazení tahů při vyhledávání ohodnocujeme tahy, ve kterých bereme soupeři figuru.

2.3.4 Generování tahů věží

Generování tahů věží je velice podobné způsobu generování tahů střelcem. Jedná se totiž o velice podobné figury. Jediný rozdíl je v tom, jakými směry se figury mohou pohybovat a ve způsobu ohodnocování tahů.

2.3.5 Generování tahů dámou

Dáma se může hýbat jako věž a střelec, jelikož generování tahů věží i generování tahů střelcem je už vytvořeno. Stačí použít již existující funkce.

2.3.6 Generování tahů králem

U generování tahů králem se nabízí implementace podobná generování tahů jezdce. Mít tedy předem definované pole, kde pro každou možnou pozici budeme mít bitovou šachovnici obsahující možné tahy. Generování se dá také udělat poměrně jednoduše pomocí bitových operací.

Při testování těchto dvou způsobů jsem došel k závěru, že způsob generování tahů pomocí bitových operací je v průměru asi o desetinu procenta rychlejší. Zvolil jsem tedy tuto metodu generování tahů.

Princip implementace je podobný jako u generování tahů střelcem.

2.3.7 Generování rošád

Rošáda je v šachu specifický tah, který s sebou nese několik pravidel. Jedno z nich je, že se nemůže s králem ani s věží hýbat před zahráním rošády. Dále nemůže být mezi králem a věží žádná figura a pole, po kterých se král pohybuje nesmí ohrožovat žádná z protivníkových figur.

```
1   //LEFT_UP
2   U64 temp = bishops;
3   U64 attacks;
4   U64 newPossition;
5   char i = 0;
6   while(temp){
7     i++;
8     temp = ((((temp & NOT_H_FILE) & (~ROW_8)) << LEFT_UP) & (~
          myPieces));
9     attacks = temp & opponentPieces;
10    temp &= ~opponentPieces;
11    while (attacks) {
12      char sq = POP(&attacks);
13      mv.from = sq - LEFT_UP * i;
14      mv.to = sq;
15      ...
16      (*moves).pushMove(mv);
17    }
18    newPossition = temp;
19    while (newPossition) {
20      char sq = POP(&newPossition);
21      mv.from = sq - LEFT_UP * i;
22      mv.to = sq;
23      ...
24      (*moves).pushMove(mv);
25    }
26  }
```

Zdrojový kód 6: Kód generování tahů střelcem

Vezměme si tedy příklad generování rošády u bílého na královském křídle. Musíme zkontrolovat, zda se král a věž nacházejí na správných pozicích. Pokud ano, tak zkontrolujeme, jestli se na polích f1 a g1 nenachází žádné figury. Nakonec zkontrolujeme, zda protivník neohrožuje pole e1 a f1. Pokud neohrožuje, je možné udělat rošádu. Čtenář si teď může myslet, že jsem zapomněl na ověření pole g1, kam se král po provedení rošády posune. Samozřejmě pokud protivník ohrožuje pole g1, tak není možné udělat rošádu. Nicméně to ověřujeme při hledání. Při zahrání libovolného tahu se kontroluje, zda tah nevystaví krále do šachu. Pokud by vystavil, je tah brán jako nevalidní.

Podobný princip generování možných rošád platí i pro ostatní směry. V kódu si můžete všimnout parametru castelingOptions. Zde se udržuje informace o tom, jaké rošády může hráč na tahu udělat. Jelikož samotná kontrola, jestli jsou figury na správném místě, nestačí. Například pokud hráč pohne králem na e2 a následně zpět na e1, už nemůže udělat rošádu.

```
1   U64 pos, attacks;
2   while (knights) {
3     char sq = POP(&knights);
4     pos = KnightOffsets[sq];
5     pos &= ~myPieces; // don't have own piece on the square
6     attacks = pos & opponentPieces;
7     pos &= ~opponentPieces;
8     while (attacks) {
9       char to = POP(&attacks);
10      mv.from = sq;
11      mv.to = to;
12      ...
13      (*moves).pushMove(mv);
14    }
15    while (pos) {
16      char to = POP(&pos);
17      mv.from = sq;
18      mv.to = to;
19      ...
20      (*moves).pushMove(mv);
21    }
22  }
```

Zdrojový kód 7: Generování tahů jezdcem

```
1  void QueenMoves(...) {
2    RookMoves(...);
3    BishopMoves(...);
4  }
```

Zdrojový kód 8: Generování tahů dámou

```
U64 pos = ((king & NOT_A_FILE) >> LEFT) | ((king & NOT_H_FILE) << LEFT) |
  ((king & (~ROW_8)) << UP) | ((king & (~ROW_1)) >> UP);
 pos |= (((king & NOT_H_FILE) & (~ROW_8)) << LEFT_UP) | (((king & NOT_A_FILE) & (~ROW_8)) << RIGHT_UP);
 pos |= (((king & NOT_A_FILE) & (~ROW_1)) >> LEFT_UP) | (((king & NOT_H_FILE) & (~ROW_1)) >> RIGHT_UP);

 pos &= ~myPieces;
 U64 capture = pos & opponentPieces;
 pos &= ~opponentPieces;

 char from = POP(&king);
 while (capture) {
  char sq = POP(&capture);
  mv.from = from;
  ...
  (*moves).pushMove(mv);
 }
 while (pos) {
  char sq = POP(&pos);
  mv.from = from;
  mv.to = sq;
  ...
  (*moves).pushMove(mv);
 }
```

Zdrojový kód 9: Generování tahů králem

```
  if (king & (1ULL << e1)) {
   if (((castelingOptions == BothSides || castelingOptions == KingSide) && (rooks & (1ULL << h1)))
    && (!((myPieces | opponentPieces) & ((1ULL << f1) | (1ULL << g1))))) {
    if (!Threathens(1ULL << e1, opponentPieceList, ...)) {
     //king side
     //check if opponent threathens f1
     if (!Threathens(1ULL << f1, opponentPieceList, ...)) {
      mv.from = e1;
      mv.to = g1;
      mv.enPassant = 0;
      mv.transforms = EMPTY;
      mv.isCastelingMove = true;
      mv.movedPiece = whiteKing;
      (*moves).pushMove(mv);
     }
    }
   }
```

Zdrojový kód 10: Generování rošády pro bílého na královském křídle

2.4 Hashování šachovnice

Při vyhledávání si ukládáme informaci o nejlepším tahu pro danou pozici do transpoziční tabulky. Následně transpoziční tabulku využíváme pro řazení tahů a ořízntutí redundantních pozic ve vyhledávacím stromu. Redundantní pozice může nastat například prohozením dvou tahů u jednoho z hráčů.

Abychom mohli transpoziční tabulku využívat, je potřeba mít hashovací funkci, která šachovnici přiřadí klíč. Následně můžeme tento klíč využít pro indexaci v transpoziční tabulce.

Klíč počítáme za pomoci náhodně vygenerovaných čísel. Pro každou figuru a každé pole, na kterém se může figura nacházet, vygenerujeme náhodné 64bitové číslo. Dále vygenerujeme 64 náhodných čísel pro braní mimochodem. Tady by stačilo pouze 16 náhodných čísel, ale režie s přepočtem indexu je příliš velká. Také potřebujeme jedno 64bitové číslo pro identifikaci hráče na tahu a 4 další čísla pro přidání informace o možnosti rošád. Klíč má 64 bitů a je na začátku vypočítán postupně aplikací operace XOR na jednotlivá čísla korespondující s pozicí figur na šachovnici. Následně při hraní tahu je klíč vždy upraven dle změny pozice figur a dalších faktorů.

```
1   void Init() {
2     for (int i = 0; i < 12; i++) {
3       for (int j = 0; j < 64; j++) {
4         PiecesRand[i][j] = getRandom64();
5       }
6     }
7     for (int j = 0; j < 64; j++) {
8       enPass[j] = getRandom64();
9     }
10    SideOnMove = getRandom64();
11    for (int i = 0; i < 4; i++) {
12      CastlePermition[i] = getRandom64();
13    }
14  }
```

Zdrojový kód 11: Inicializace konstant pro hashování

Při inicializaci šachovnice spočítáme první klíč. Následně klíč upravujeme při zahrání tahu. Při hashování klasického tahu odebereme z klíče pomocí operace XOR předchozí umístění figury. Naopak přidáme konečnou pozici figury. Nesmíme zapomenout odebrat branou figuru a upravit klíč na základě hodnot braní mimochodem. V neposlední řadě potom upravíme informaci o hráči na tahu. Pokud tah promění figuru na jinou, je třeba přidávat do klíče hodnotu z pole, které koresponduje s nově nasazenou figurou. Pro braní mimochodem musíme rozlišit bílého a černého hráče.

Pro hashování rošády je opět potřeba mít speciální funkci. Níže je ukázka kódu pro hashování rošády bílého na dámském křídle. Ostatní strany jsou implementovány podobným způsobem. Nejen rošády, ale také ostatní tahy mohou

```
unsigned long long defaultHash(Bitboard *b) {
  U64 temp = 0;
  unsigned long long result = 0;
  for (int i = 0; i < 6; i++) {
    temp = b->whitePiecesList[i];
    while (temp) {
      result ^= PiecesRand[i][POP(&temp)];
    }
    temp = b->blackPiecesList[i];
    while (temp) {
      result ^= PiecesRand[i + 6][POP(&temp)];
    }
  }

  if (b->whiteOnMove) {
    result ^= SideOnMove;
  }


  if (b->enPassant != 0) {
    result ^= enPass[getEnPassantIndex(b->enPassant)];
  }

  result ^= CastlePermition[(*b).whiteCastleAlowed];
  result ^= CastlePermition[(*b).blackCastleAlowed];

  return result;
}
```

Zdrojový kód 12: Hashovací funkce používaná při inicializaci šachovnice

změnit přípustné rošády. Proto je implementována další funkce, která řeší úpravu klíče na základě změny možností rošád.

```
unsigned long long hashMove(U64 key, Move* mv) {
  //Remove Old hash
  key ^= PiecesRand[mv->movedPiece - 1][mv->from];
  //Add moved
  key ^= PiecesRand[mv->movedPiece - 1][mv->to];
  // Remove taken
  if (mv->taken) {
    key ^= PiecesRand[mv->taken - 1][mv->to];
  }
  if (mv->enPassant) {
    key ^= enPass[getEnPassantIndex(mv->enPassant)];
  }
  if (mv->enPassantBeforeMove) {
    key ^= enPass[getEnPassantIndex(mv->enPassantBeforeMove)];
  }

  //Hash side
  key ^= SideOnMove;

  return key;
}
```

Zdrojový kód 13: Funkce pro hashování tahu

```
unsigned long long hashWhiteEnPassantMove(U64 key, Move* mv) {
  //Remove Old hash
  key ^= PiecesRand[mv->movedPiece - 1][mv->from];
  //Add moved
  key ^= PiecesRand[mv->movedPiece - 1][mv->to];
  // Remove taken
  key ^= PiecesRand[mv->taken - 1][mv->to - 8];

  if (mv->enPassant) {
    key ^= enPass[getEnPassantIndex(mv->enPassant)];
  }
  if (mv->enPassantBeforeMove) {
    key ^= enPass[getEnPassantIndex(mv->enPassantBeforeMove)];
  }

  //Hash side
  key ^= SideOnMove;
  return key;
}
```

Zdrojový kód 14: Hashování bílého enPassant tahu

```
unsigned long long hashCastelingQueenWhiteMove(U64 key, Move * mv) {
  //Remove Old hash
  key ^= PiecesRand[5][e1];
  key ^= PiecesRand[3][a1];
  //Add moved
  key ^= PiecesRand[5][c1];
  key ^= PiecesRand[3][d1];

  //Hash side
  key ^= SideOnMove;

  if (mv->enPassant) {
    key ^= enPass[getEnPassantIndex(mv->enPassant)];
  }
  if (mv->enPassantBeforeMove) {
    key ^= enPass[getEnPassantIndex(mv->enPassantBeforeMove)];
  }
  return key;
}
```

Zdrojový kód 15: Hashování rošády bílého na dámském křídle

```
unsigned long long hashCastelingChange(U64 key, char from, char to)
    {
  //Remove Old hash
  key ^= CastlePermition[from];
  key ^= CastlePermition[to];
  return key;
}
```

Zdrojový kód 16: Úprava klíče na základě změny možností rošád

2.5 Hraní tahů

Tahy jsou již vygenerovány, nyní je potřeba mít metody pro hraní tahu a braní tahu zpět. U těchto metod jsem se snažil co nejvíce využít binárních operací, především z důvodu rychlosti.

2.5.1 Metoda PlayMove

Nejdříve se podíváme na metodu, která má za úkol zahrát tah. Metoda bere jako argument tah, který má zahrát. Navíc používá atributy bitové šachovnice, kterou má uloženou v klíčovém slově this.

První krok, který je potřeba při hraní tahu udělat, je zapamatovat si momentální nastavení šachovnice. Zahraný tah budeme s největší pravděpodobností vracet zpět. Je tedy potřeba uložit stav bitové šachovnice před zahráním tahu.

Ukládat všechny atributy není nutné. Bylo by to neefektivní a zbytečně pomalé. Stačí nám uložit pouze několik vybraných atributů tak, aby se dal předchozí stav odvodit ze struktury tahu. Atributy, které přidáváme k tahu jsou momentální rošádové možnosti pro obě strany a enPassant pole. Při braní tahu zpět totiž není možné tyto hodnoty odhadnout pouze na základě tahu.

Následně probíhá samotné hraní tahu a přepočítávání hashovacího klíče po zahrání tahu. Nejdříve se podíváme na to, zda tah změní možnosti rošády pro hráče na tahu. Pokud ano, je změna zahashovaná.

Samotné zahrání tahu se dá potom typicky rozdělit do tří kroků:

1. Odeber figuru, se kterou taháš.
2. Odeber branou figuru.
3. Přidej figuru na místo tahu.

U braní mimochodem nastavíme tahu příznak enPassant, aby bylo jednodušší tah vzít zpět. Po provedení libovolného tahu je potřeba upravit klíč šachovnice. Pro jednoduchost by se dal klíč přepočítat úplně celý znovu, nicméně mnohem rychlejší a efektivnější řešení je vzít původní klíč a upravit ho podle zahraného tahu. Pochopitelně u braní mimochodem se klíč upravuje jiným způsobem než při hraní normálního tahu, jak už se čtenář dozvěděl v předchozí kapitole.

Dále je třeba rozlišovat různé speciální tahy. Jako první se podíváme na braní mimochodem. O braní mimochodem jde, pokud je pole, kam se pohybuji, rovno enPassant poli a zároveň pohnu pěšcem.

Další speciální tah je poté transformace figury, když například bílý pěšec dojde na poslední řadu. Zde opět používáme stejný princip. Nejdříve tedy odebereme pěšce, se kterým táhneme. Následně odebereme branou figuru, pokud taková existuje. Následně přidáme novou figuru na pozici tahu. Používáme při tom atributy taken a transforms, které byly nastaveny při generování tahu.

Poslední speciální tah je potom rošáda. Zde je princip opět stejný, jen se v principu jedná o dva tahy, jeden tah králem a druhý tah věží. Následně je potřeba upravit klíč po zahrání tahu.

```
1     if (((1ULL << (*mv).to) & enPassant) && (*mv).movedPiece ==
          whitePawn ) {
2       whitePiecesList[0] &= ~(1ULL << mv->from); // remove old piece
3       whitePieces &= ~(1ULL << mv->from); // remove old piece
4
5       blackPiecesList[0] &= ~((1ULL << mv->to) >> UP); // remove
            taken piece
6       blackPieces &= ~((1ULL << mv->to) >> UP);
7
8       mv->isEnPassantMove = true;
9       whitePiecesList[0] |= (1ULL << mv->to); // add new piece
10      whitePieces |= (1ULL << mv->to); // add new piece
11
12      mv->keyAfterMove = hashWhiteEnPassantMove(mv->keyAfterMove, mv)
            ;
13    }
```

Zdrojový kód 17: Hraní tahu: braní mimochodem

Po provedení tahu je potřeba nastavit několik atributů. Nejdříve potřebujeme upravit atribut indikující počet tahů bez pohybu pěšce a vzetí figury. Následně je potřeba přidat zahraný tah do historie a upravit enPassant pole. Metoda jako výsledek vrací indikátor určující, jestli byl tah validní. Pokud by totiž byl po zahrání tahu král v ohrožení, není možné tento tah zahrát.

```
1     else if ((*mv).transforms != EMPTY) {
2       //doTransformationMove(mv);
3       whitePiecesList[0] &= ~(1ULL << mv->from); // remove old piece
4       whitePieces &= ~(1ULL << mv->from); // remove old piece
5       //blackPiecesList[0] &= ~((1ULL << mv->to)); // remove taken
            piece if there is any
6
7       if (mv->taken) {
8         blackPieces &= ~(1ULL << mv->to);
9         blackPiecesList[mv->taken - 7] &= ~(1ULL << mv->to);
10      }
11
12      whitePiecesList[mv->transforms - whitePawn] |= (1ULL << mv->to)
            ; // add new piece
13      whitePieces |= (1ULL << mv->to); // add new piece
14
15      mv->keyAfterMove = hashTransformationMove(mv->keyAfterMove, mv)
            ;
16    }
```

Zdrojový kód 18: Hraní tahu: transformace figury

```
1     else if ((*mv).isCastelingMove) {
2       if (mv->to == c1) {
3         //queenside
4         whitePiecesList[5] &= ~(1ULL << mv->from); // remove old piece
5         whitePiecesList[5] |= (1ULL << mv->to); // add new piece
6         whitePiecesList[3] &= ~(1ULL << a1); // remove old piece
7         whitePiecesList[3] |= (1ULL << d1); // add new piece
8         whitePieces &= ~((1ULL << mv->from) | (1ULL << a1)); // remove
              old piece
9         whitePieces |= ((1ULL << mv->to) | (1ULL << d1)); // add new
              piece
10        mv->keyAfterMove = hashCastelingQueenWhiteMove(mv->
              keyAfterMove, mv);
11      }
```

Zdrojový kód 19: Hraní tahu: rošáda dámské křídlo

2.5.2 Metoda UndoMove

Při braní tahu zpět používá engine historii tahů, do které přidává zahrané tahy. Když chceme tah vzít zpět, funkce se podívá na poslední zahraný tah a na základě jeho atributů vrátí šachovnici do předchozího stavu.

Předtím než se budou nastavovat figury, nastavíme atributy šachovnice, které jsme si uložili při hraní tahu. Jedná se o enPassant pole, rošádové možnosti a počet tahů bez vzetí figury nebo tahu pěšcem.

Následně se postaráme o správné nastavení figur na šachovnici. Používáme při tom opačný princip než při hraní tahu. Nejdříve tedy vezmeme figuru z pozice, kam jsme táhli a vrátíme ji na původní pozici. Je také potřeba vrátit soupeřovu figuru, pokud jsme nějakou brali.

Opět je zde několik speciálních tahů. Nejdříve tah braní mimochodem, kde nevracíme soupeřovu figuru na místo, kam jsme táhli. Další speciální tah je proměna pěšce, kde je potřeba odebrat správný typ figury ze šachovnice a poslední speciální tah je rošáda, kde používáme stejný princip jako pro tah, nicméně se jedná o hraní dvou tahů naráz, jeden tah králem a druhý věží.

```
1     if (last.isEnPassantMove) {
2       this->whitePiecesList[0] |= (1ULL << last.from);
3       this->whitePiecesList[0] &= ~(1ULL << last.to);
4       this->whitePieces |= (1ULL << last.from);
5       this->whitePieces &= ~(1ULL << last.to);
6
7       this->blackPiecesList[0] |= ((1ULL << last.to) >> UP); // >>
            because from white view we are goig down
8       this->blackPieces |= ((1ULL << last.to) >> UP);
9     }
```

Zdrojový kód 20: Braní tahu zpět: braní mimochodem

Funkce odebírání tahu nevrací žádnou hodnotu, jelikož není potřeba.

2.6 Generování bracích tahů

Při tvorbě stromu prohledávání stavového prostoru se snažíme předejít takzvanému horizon efektu. Horizon efekt znamená, že bychom ohodnocovali pozici, ve které by existoval tah, který by mohl zásadně změnit ohodnocení. Například, že by hráč vzal protivníkovi dámu. Tomuto se dá předejít tím, že budeme prohledávat pouze brací tahy, dokud se nedostaneme do klidné pozice. Pozice je klidná, pokud hráč na tahu nemůže vzít protivníkovi žádnou figuru.

Pro to tedy potřebujeme funkce pro generování bracích tahů. Generování těchto tahů je prakticky totožné jako generování všech tahů. Nicméně teď ignorujeme normální tahy a přidáváme pouze tahy, které berou protivníkovu figuru.

Zde je jednoduchá ukázka kódu pro generování bracích tahů pro střelce. Čtenář si může odvodit, jak se generují tahy pro ostatní figury.

```
1   //LEFT_UP
2   U64 temp = bishops;
3   U64 attacks;
4   char i = 0;
5   while (temp) {
6     i++;
7     temp = ((((temp & NOT_H_FILE) & (~ROW_8)) << LEFT_UP) & (~
          myPieces));
8     attacks = temp & opponentPieces;
9     temp &= ~opponentPieces;
10    while (attacks) {
11      char sq = POP(&attacks);
12      mv.from = sq - LEFT_UP * i;
13      mv.to = sq;
14      mv.enPassant = 0;
15      mv.transforms = EMPTY;
16      for (int j = 0; j < 5; j++) {
17        if (opponentPieceList[i] & (1ULL << sq)) {
18          mv.score = value[i] - BISHOP_VAL;
19          mv.taken = base + i; // piece - 2 because bishop and pawn
20          break;
21        }
22      }
23      mv.movedPiece = piece;
24      (*moves).pushMove(mv);
25    }
26  }
```

Zdrojový kód 21: Generování bracích tahů střelcem

2.7 Počítání možných tahů

Když se snažíme vypočítat, který hráč má lepší pozici, je potřeba vzít v úvahu počet přípustných tahů, které hráč může zahrát. Typicky hráč, který může zahrát

více tahů, má větší mobilitu a tím pádem i výhodu.

K výpočtu počtu tahů nám slouží funkce, které jsou velice podobné funkcím generujícím tahy. Jedná se v podstatě o stejný princip, jen nevytváříme tahy, ale počítáme přípustná pole.

3 Null Move Pruning

3.1 Řazení tahů

Řazení tahů je velice důležité při procházení vyhledávacího stromu. Když procházíme tahy, tak nejdříve vybereme tah, který byl uložen jako nejlepší v předchozím vyhledávání. Nicméně typicky procházíme více tahů v jedné pozici, proto je důležité mít tahy seřazené od nejlepšího po nejhorší.

Ohodnocení tahů je ale složitější úkol, než se na první pohled může zdát. Máme totiž pouze minimální informace o tom, jak je tah dobrý. Přesnost ohodnocení je úměrná výpočetnímu času. S rostoucí přesností roste i čas, který engine stráví ohodnocením tahu. Je nutné mít na paměti, že v typické pozici engine projde pouze několik prvních tahů. Ostatní jsou ořezány. Ohodnocení tahů je tedy prováděno velice primitivně ale rychle. Tah, který bere soupeřovi figuru, je většinou lepší než normální tah. Proto je takový tah třeba priorizovat. Dále je třeba uvažovat, jakou figuru protivníkovi bereme.

Při testování řazení tahů jsem zkoušel přidat konstanty pro figuru, kterou bereme tak, aby engine preferoval braní pěšcem například před braní dámou. Ohodnocení tahu bylo rovno hodnotě brané figury mínus hodnota figury, kterou bereme. Viz konstanty níže.

```
1 const short pieceValue[] = {
     100,300,300,500,900,900,6000,100,300,300,500,900,6000 };
2 const short PAWN_VAL = 9;
3 const short BISHOP_VAL = 29;
4 const short KNIGHT_VAL = 29;
5 const short ROOK_VAL = 49;
6 const short QUEEN_VAL = 89;
7 const short KING_VAL = 89;
```

Zdrojový kód 22: Konstanty pro ohodnocení figur

Při testování jsem ale přišel na to, že je lepší nebrat v úvahu to, jakou figurou bereme. Jednoduše to nevypovídá tolik o tom, jak je tah kvalitní. Výsledek testování můžete vidět na tabulce níže.

Verze	Výhry	Prohry	Remízy
Engine ohodnocující brací figuru	2	9	3
Engine ignorující jakou figurou bereme	9	2	3

Engine, který byl implementovaný v bakalářské práci, používá minimax vyhledávání s alphabeta ořezáváním a late move reduction, kde late move reduction spočívá v prohledávání horších tahů do zredukované hloubky. Jako další vylepšení enginu jsem implementoval Null Move Pruning.

3.2 Co je null move pruning?

Oproti ostatním vylepšením šachového enginu není null move reduction odvozené od způsobu, jakým hrají šach lidé. Například late move reduction je pěkný příklad, kdy se snažíme napodobit myšlení člověka při hraní šachu. Naopak null move pruning nemá nic společného s tím, jak hrají šach lidé. Null move pruning by se dal vysvětlit analogií z boxu, kde dáte protihráči možnost vás udeřit. Pokud ránu vydržíte, pravděpodobně vás protihráč nedokáže porazit. V implementaci šachového enginu to potom znamená, že dáte protivníkovi možnost zahrát tah navíc. Jinými slovy nezahrajete tah.

3.3 Princip implementace

V šachové pozici je typicky nejhorší možnost nehrát žádný tah. I když to pravidla šachu neumožňují, můžeme tento fakt využít při zlepšení šachového enginu. Předpokládejme tedy, že když nezahrajeme žádný tah a provedeme vyhledávání do zredukované hloubky, tak dostaneme nejhorší možné ohodnocení šachovnice v daném uzlu, což samozřejmě neplatí pro každou pozici v šachu, existují pozice, ve kterých by byla možnost nezahrát tah nejlepší. Takových pozic je ale velice málo a nemá cenu je brát v úvahu.

Pokud je ohodnocení zredukovaného vyhledávání po provedení null move větší nebo roven momentální horní hranici vyhledávání beta, můžeme větev ořezat vrácením hodnoty beta jako výsledek prohledávání dané pozice. Null move prunning není možné použít v každé pozici. Například jej nelze provést, pokud je král hráče, který je na tahu, ohrožen. Dále není vhodné provádět null move pruning, pokud je hloubka vyhledávání menší nebo rovna hodnotě, kterou redukujeme hloubku vyhledávání. Tato hodnota je často označována v literatuře jako R.

Dále ze zjevných důvodů není možné provést dva null tahy po sobě. A také nemůžeme provést null move pruning v prvním zavolání hledání, jelikož bychom nemuseli obdržet žádný tah.

3.4 Výsledek implementace

V tabulce níže je popsán výsledek přidání null move pruning do šachového enginu. Zajímavé je především to, jak velký vliv může mít null move pruning na hloubku hledání a tím i na výkonnost enginu.

Verze	Výhry	Prohry	Remízy
Bez null move	0	7	4
S null move	7	0	4

```
1  if (nullPruning) {
2     if (whiteOnMove) {
3       U64 kingPos = bitboard->whitePiecesList[5];
4       if (depthLeft >= 4 && (!Threathens(bitboard->whitePiecesList
            [5], ...))){
5         //s->maxDepthQuiesce = - (depthLeft - 4);
6         U64 enPass = bitboard->enPassant;
7         bitboard->enPassant = 0;
8         score = -alphaBeta(s, -beta, -beta + 1, depthLeft - 4, !
              whiteOnMove, false, bitboard);
9         bitboard->enPassant = enPass;
10        bitboard->whiteOnMove = whiteOnMove;
11        if (score >= beta) {
12          return beta;
13        }
14       }
15     }
```

Zdrojový kód 23: Null move pruning

4 Paralelní vyhledávání

Paralelizace vyhledávání může teoreticky zlepšit výkon enginu, pokud poběží na moderním procesoru. Engine bude paralelně prohledávat více pozic naráz a tím pádem bude i výkonnější. Myšlenka je tedy velice jednoduchá, mít několik vláken, která budou paralelně pracovat na vyhledávacím stromu. Je ale potřeba ošetřit několik problémů, které implementace může přinést.

4.1 Možné algoritmy pro paralelizaci vyhledávání

4.1.1 Principal Variation Splitting (PVS)

Tento algoritmus je velice jednoduchý, hlavní vlákno prochází nejlevější cestu stromu (také známou jako Principal Variation), dokud nenarazí na list. Jakmile narazí na list, ohodnotí jej a vrátí hodnotu rodiči. Na cestě od listu zpět ke kořenu hlavní vlákno upraví hodnotu alpha a následně distribuuje zbývající tahy mezi nečinná vlákna. Je důležité si všimnout, že paralelní hledání nastává až po tom, co projdeme nejlevější cestu až k listu.

Jakmile všechna vlákna dokončí hledání pro daný uzel, vrátí výsledek hlavnímu vláknu, to následně zjistí nejlepší ohodnocení a nejlepší tah. Následně může přejít k rodiči, kde celý proces opakuje, dokud se nedostane ke kořenu.

Tento přístup má problém se škálováním. Jelikož paralelně prohledává pouze jeden uzel neboli jednu pozici, existuje pouze omezený počet vláken, které využije. V průměrné pozici lze zahrát 35 tahů. Kdybychom měli více vláken, zůstaly by některé nevyužity.

Další problém je, že každé vlákno dostává jeden tah na prozkoumání. Často

se ale může stát, že jeden tah vede k mnohem složitějšímu podstromu než jiný. Tím pádem musejí ostatní vlákna čekat, dokud není dokončeno vyhledávání v nejsložitějším podstromu.

4.1.2 Young Brothers Wait Concept (YBWC)

Tento algoritmus paralelního prohledávání stromu je velice podobný výše popsanému PVS algoritmu. YBWC algoritmus je ještě striktnější než PVS. U PVS se nejdříve prohledávala nejlevější cesta a následně se zbytek prohledávání prováděl paralelně. U YBWC se prohledává nejlevější cesta jako první v každém uzlu a není možné paralelně prohledávat, dokud není dokončeno procházení nejlevějšího potomka. Následně je zbytek práce distribuován mezi ostatní uzly pomocí "kradení" práce. Jakmile uzel dokončí procházení nejlevějšího uzlu, přidá zbývající uzly do fronty. Nečinná vlákna potom mohou brát uzly z fronty.

Tím pádem odpadá problém se synchronizací, kde u PVS vlákna musela čekat, dokud není uzel plně prohledán. U YBWC si vlákna berou práce nezávisle na sobě.

4.1.3 Lazy SMP

Jedná se o nejnovější z algoritmů, kde vlákna mají minimální komunikaci mezi sebou. Jediné místo, které spolu sdílí, je transpoziční tabulka. Nápad za tímto algoritmem je až komicky jednoduchý. Nechat vlákna prohledávat strom pomocí klasického vyhledávání, které využívá transpoziční tabulku.

Největším benefitem tohoto přístupu je ten, že odpadá problém se synchronizací vláken, což s sebou nese značný overhead. Transpoziční tabulka se stará o to, že uzly, které už jedno vlákno plně prošlo, nebudou znovu procházeny.

Jediný overhead, který s sebou Lazy SMP přináší je ten, že stejná pozice může být procházena vícekrát. V nejhorším případě ji projdou všechna vlákna naráz. Jelikož neexistuje žádná komunikace mezi vlákny, vlákna neví, jestli jiné vlákno nezačalo pracovat na dané pozici. To vede k redundanci při vyhledávání, kdy více vláken provádí stejné výpočty.

Této redundanci by se dalo vyhnout, kdyby každé vlákno hledalo do jiné hloubky. Tohle ale způsobí úplně jiný problém. Vlákna hledající do vyšší hloubky nebudou mít možnost seřadit tahy na základě toho, jaký tah byl nejlepší při předchozím vyhledávání. To je velký problém, který by způsobil, že engine bude procházet tahy ve špatném pořadí.

Dokonce momentálně nejlepší šachový engine Stockfish ve své verzi 7 používá právě Lazy SMP. Rozhodl jsem se tedy implementovat Lazy SMP do enginu.

4.2 Testování

Paralelní procházení stromu s sebou nese určitý overhead. V prvních dvou algoritmech se jednalo především o komunikační a synchronizační overhead. V posledním je to spíše redundance procházení. Výsledky testování jsou nečekané.

Pravděpodobně se budou lišit zařízením, na kterém je engine spuštěn. Nicméně na mém osobním notebooku měl engine s paralelním vyhledáváním při použití 4 vláken horší výsledky než engine používající 2 vlákna.

Počet uzlů, které engine projde při použití 4 vláken je větší než počet uzlů projitých enginem používajícím 2 vlákna. Nicméně hloubka vyhledávání je u enginu používající 2 vlákna zpravidla větší. To naznačuje, že redundance procházení pozic je velký problém při implementaci Lazy SMP.

Při testování enginu s paralelním vyhledáváním s 2 vlákny proti starší verzi enginu jsem přišel na to, že vliv na výkon enginu je zanedbatelný, jak si čtenář může všimnout na tabulce níže.

Verze	Výhry	Prohry	Remízy
Předchozí verze	4	4	8
Paralelní vyhledávání	4	4	8

Vytvořil jsem tedy verzi enginu, kde vlákna hledají do různých hloubek. Při vyhledávání do hloubky i polovina vláken hledá do hloubky i + 1 a zbývající polovina do hloubky i + 2. Cílem je zamezení redundanci při vyhledávání.

Při hledání jsem používal 4 vlákna a k mému překvapení byla verze lepší než předchozí verze, kde vlákna hledala do stejné hloubky. Očividně problém s řazením tahu není až tak drastický. Nicméně proti starší verzi enginu si verze s paralelním hledáním do různých hloubek vedla hůře, než verze s paralelním hledáním do stejné hloubky.

Verze	Výhry	Prohry	Remízy
Předchozí verze	9	7	7
Paralelní vyhledávání různá hloubka	7	9	7

Verze	Výhry	Prohry	Remízy
Paralelní vyhledávání stejná hloubka	4	6	13
Paralelní vyhledávání různá hloubka	6	4	13

Problém je tedy známý, redundance procházení uzlů, které ještě nebyly plně prohledány. Za momentální implementace engine kontroluje na začátku prohledávání libovolného uzlu, zda již nebyl prohledán. Jinými slovy podívá se, jestli pro danou pozici neexistuje záznam v transpoziční tabulce s hloubkou hledání větší nebo rovné momentální hloubce hledání. Pokud záznam ale nenajde, pokračuje v prohledávání. Když následně některé z vláken dokončí práci a přidá záznam do transpoziční tabulky, ostatní vlákna právě prohledávající stejný uzel nemají možnost se o této skutečnosti dozvědět.

Jednoduché řešení, které mě napadlo, je tedy přidat více kontrol transpoziční tabulky při prohledávání uzlu. Nebudeme tedy kontrolovat jen na začátku hledání ale i v průběhu hledání. Přesněji při procházení jednotlivých tahů. Každé vlákno tedy u třetího a osmého tahu bude navíc kontrolovat transpoziční tabulku a pokud v ní existuje záznam se stejnou hloubkou, ví, že jiné vlákno již dokončilo prohlížení tohoto uzlu.

Tato funkcionalita s sebou pochopitelně nese určitý overhead ve formě kontroly transpoziční tabulky. To se také podepsalo na výsledném výkonu enginu. Níže je uvedena tabulka popisující jak engine s kontrolou ukončení hrál proti jiným verzím enginu. Počet výher, remíz a proher je v tabulce uveden z pohledu testovaného enginu. Čili dle prvního řádku engine 1 krát vyhrál nad enginem se sekvenčním vyhledáváním, 1 krát remizoval a 11 krát prohrál.

Verze	Výhry	Prohry	Remízy
Sekvenční vyhledávání	1	11	1
Paralelní vyhledávání stejná hloubka	0	11	3
Paralelní vyhledávání různá hloubka	0	11	3

4.3 Závěr

Po testování paralelního vyhledávání jsem se rozhodl tuto funkcionalitu do enginu nepřidávat. Bylo by zajímavé otestovat, jaký vliv má to, na jakém procesoru engine běží. Je možné, že kdyby engine běžel na výkonnějším procesoru, byl by výsledek paralelního vyhledávání úplně jiný.

5 Ukládání ohodnocení šachovnice

5.1 Paralelní ohodnocení šachovnice

Při ohodnocování šachovnice bereme v potaz několik faktorů jako například pozice figur, bezpečnost krále a tak dále. Bylo by tedy možné použít vlákna pro hodnocení jednotlivých kritérií paralelně a následně potom pouze sečíst výsledky jednotlivých faktorů.

Implementace je tedy velice jednoduchá, stačí vytvořit několik vláken tak, aby každé vlákno zpracovávalo jeden faktor ohodnocení. Vlákna ovšem nejsou zdarma, mají spolu spojený určitý overhead. A jelikož jednotlivé faktory na ohodnocení při použití bitových šachovnic nejsou výpočetně složité, implementace paralelního ohodnocování šachovnice nepřineslo žádné benefity. Právě naopak, implementace s paralelním ohodnocením byla výkonnostně slabší než implementace s klasickým výpočtem ohodnocení šachovnice.

5.2 Ukládání ohodnocení šachovnice

Typicky při vývoji softwaru se dá zlepšit výkonnost opakujících se výpočtů pomocí cachingu. Jednoduše si uložíme výsledek z předchozího výpočtu a následně, když jsme dotazování po novém výpočtu, podíváme se do uložených výsledků. Pointa tohoto principu je, že většinou se nám vyplatí podívat se do paměti a zkontrolovat, zda jsme danou pozici již neprocházeli.

Stejný princip funguje u transpoziční tabulky, která je popsána v bakalářské práci. Otázka, na kterou jsem se snažil najít odpověď, tedy zní: Dá se použít stejný princip transpoziční tabulky i na ohodnocení šachovnice?

5.3 Implementace

Implementace je poněkud jednoduchá, stačí nám pár řádků kódu. Nejdříve si alokujeme pole určité velikosti. Následně si inicializujeme hodnoty tohoto pole na nereálnou hodnotu ohodnocení šachovnice. Při zavolání funkce rateBoard se podíváme, zda náhodou není pro daný klíč šachovnice záznam v naší tabulce. Pokud je, vrátíme hodnotu z tabulky. Zde je ještě třeba zmínit, že ohodnocení se do tabulky ukládá z pohledu bílého. Je tedy ještě potřeba rozlišit, jaký hráč je na tahu a upravit znaménko ohodnocení. Pokud nenajdeme záznam v tabulce, tak se provede klasické ohodnocení. Na konci uložíme záznam do tabulky.

5.4 Rizika

Jelikož hashování šachovnice není perfektní, jinými slovy dochází ke kolizím. Může dojít i ke kolizi při zjišťování hodnoty z tabulky ohodnocení. Navíc při výpočtu indexu používáme operaci modulo. Ke kolizím tedy v určitém bodě dojde. To ovšem může vést k úplně nesmyslným tahům, kdy si engine myslí, že dokáže dostat skvělé ohodnocení díky kolizi v tabulce ohodnocení.

```
1 const int tableSize = 1024 * 1024 * 40;
2 int *rateTable = (int*)malloc(tableSize * sizeof(int));
3 void initTable() {
4   for (int i = 0; i < tableSize; i++) {
5     rateTable[i] = -INFINITE;
6   }
7 }
```

Zdrojový kód 24: Inicializace tabulky

5.5 Výsledek

Pro testování jsem zvolil tabulku o velikosti 1024 * 1024 *1024, to mělo za následek, že engine využíval paměť 4GB. Zde je ovšem započítána i Transpoziční tabulka a další funkce.

Intuitivně se může zdát, že implementace takové tabulky zrychlí ohodnocení. Pokud záznam v tabulce nenajdeme, budeme platit pouze velice malou daň co se výpočtu týče. Naopak pokud záznam v tabulce je, můžeme si ušetřit celé ohodnocení šachovnice.

Na tabulce níže jsou popsány výsledky používající při testování enginu ukládání ohodnocení proti předchozí verzi, která tuto funkcionalitu neobsahovala. Jak si čtenář může všimnout, rizika popsaná v této kapitole se bohužel naplnila. V enginu po několika tazích docházelo ke kolizím v ohodnocovací tabulce, to vedlo k nesmyslným tahům a to vedlo k velice špatnému výsledku.

Verze	Výhry	Prohry	Remízy
Předchozí verze	11	0	0
Ukládání ohodnocení	0	11	0

6 Razoring

Narozdíl od alpha beta ořezávání vyhledávacího stromu razoring ořezává uzly dopředu. Při alpha beta nastane ořezávání až při procházení uzlu, zatímco u razoring se snažíme zjistit, zda je vlastně vůbec potřeba tuto pozici procházet ještě před tím, než zahrajeme libovolný tah.

Razoring funguje na podobném principu jako null move pruning. Pokud je momentální ohodnocení pozice menší než alpha, tak předpokládáme, že protivník bude schopen najít alespoň jeden tah, kterým pozici vylepší.

6.1 Verze razoringu

Existuje několik verzí razoringu. Některé se snaží ohodnotit pozici a následně ji porovnat s hodnotou beta a pokud je větší, vrátí betu a neprocházejí tento uzel. Jiné se snaží použít evaluaci jen pro snížení hloubky hledání.

To jsou zajímavé přístupy, které jednoznačně stojí za to otestovat.

6.2 Implementace

Při implementaci jsem zkoušel použít různé verze razoringu, abych je následně mohl otestovat proti sobě. První verze je implementována následovně:

```
1   if (rateBoard(bitboard, true) + 200 < beta) {
2     int value = Quiesce(s, alpha, beta, - 1, whiteOnMove, bitboard);
3     if (value < beta) {
4       return value;
5     }
6   }
```

Zdrojový kód 25: Razoring první verze

Zde používáme margin 200 bodů ohodnocení, což je rovno dvěma pěšcům. Razoring probíhá, pokud jsou podmínky null move pruning splněny. Můžete si všimnout, že používáme také Quiesce vyhledávání, které prohledává pouze brací tahy, pro ověření správnosti ohodnocení.

Výsledky při porovnání s předchozí verzí jsou zobrazeny v tabulce níže.

Verze	Výhry	Prohry	Remízy
Předchozí verze	11	4	10
Razoring	4	11	10

Druhá verze používá jednodušší algoritmus snížení hloubky. Pokud se při vyhledávání dostaneme do hloubky 4 a ohodnocení pozice plus margin je menší nebo rovna alpha hodnotě, snížíme hloubku hledání na 1.

```
1   if (depthLeft == 4 && (rateBoard(bitboard, true) + 200 <= alpha))
        {
2     depthLeft = 1;
3   }
```

Zdrojový kód 26: Razoring druhá verze

Výsledek při porovnání s předchozí verzí

Verze	Výhry	Prohry	Remízy
Předchozí verze	47	39	14
Razoring	39	47	14

Poslední třetí verze je podobná druhé verzi, kde ovšem snížení hloubky vyhledávání není až tak drastické. U každého uzlu při zbývající hloubce hledání 2 provedeme podobný kód, jako je uvedeno výše. Nicméně teď snižujeme zbývající hloubku vyhledávání na nulu. Jinými slovy provedeme pouze Quiescence vyhledávání.

```
1   if (depthLeft == 2 && (rateBoard(bitboard, true) + 200 <= alpha))
        {
2     depthLeft = 0;
3   }
```

Zdrojový kód 27: Razoring třetí verze

Výsledek při porovnání s předchozí verzí je zobrazen na tabulce níže. Jde vidět, že se jedná o nejlepší verzi Razoringu dle výsledků proti předchozí verzi enginu, kde jako jediná má tato verze kladnou bilanci.

Verze	Výhry	Prohry	Remízy
Předchozí verze	56	58	28
Razoring	58	56	28

Fakt, že má nová verze enginu lepší bilanci oproti starší verzi mě vedlo k otestování obou verzí proti jinému enginu. Pro tyto účely jsem zvolil engine Cinnamon, který má elo 2112 bodů. Na tabulce níže si ale může čtenář všimnout, že výsledky předchozí verze jsou značně lepší než výsledky verze enginu s razoringem. Rozhodl jsem se tedy tuto funkcionalitu do enginu nepřidávat.

Verze	Výhry	Prohry	Remízy
Předchozí verze	6	6	2
Cinnamon	6	6	2

Verze	Výhry	Prohry	Remízy
Razoring	1	9	3
Cinnamon	9	1	3

7 Monte Carlo vyhledávání

Další zajímavá metoda prohledávání hracího stromu je Monte Carlo metoda. Monte Carlo vyhledávání se používá například v šachovém enginu od Googlu s názvem AlphaZero. Nicméně AlphaZero co se týče architektury enginu není klasický šachový engine. AlphaZero se spoléhá především na neuronové sítě, zatímco ostatní top enginy se soustředí spíše na optimalizaci klasického alpha beta vyhledávání.

To mě vedlo k otázce: je Monte Carlo metoda vyhledávání až tak špatná a pokud ano, jak velký rozdíl je mezi použitím minimax a Monte Carlo vyhledáváním?

7.1 Co je metoda Monte Carlo?

Metoda Monte Carlo je způsob procházení hracího stromu. Při minimax metodě procházíme strom do určité hloubky, v listových uzlech ohodnotíme pozici a při propagaci hodnoty nahoru ke kořenu zjistíme nejlepší tah pro danou pozici. U Monte Carlo je způsob zjištění nejlepšího tahu jiný. Monte Carlo vyhledávání se skládá ze 4 částí.

1. Selection, česky výběr - prochází se strom od kořene až k listovému uzlu
2. Expansion - přidáme nového potomka k uzlu, který jsme vybrali v první fázi
3. Simulace - hrajeme náhodné tahy, dokud někdo nevyhraje nebo nenastane remíza
4. Backpropagation - propagujeme výsledek nahoru stromem

Nejzajímavější fáze je selection. Existuje spousta způsobů, pomocí kterých můžeme vybírat uzly, které budeme procházet. Nejjednodušší z hlediska implementace je procházet uzly náhodně. Jinými slovy v každé pozici náhodně vybrat tah, který budeme procházet dále. Tento způsob je sice jednoduchý na implementaci, ale není vhodný pro hru se spoustou možností jako jsou šachy.

Lepší přístup by byla snaha najít rovnováhu mezi procházením nových uzlů a procházením uzlů, které vypadají jako nejlepší možné tahy na základě dosavadního vyhledávání. Typicky uzel, který vede k 50 výhrám, bude lepší než uzel, který vede k 50 prohrám.

Další zajímavá fáze je simulace, kde dle základního Monte Carlo algoritmu hrajeme, dokud nedostaneme výsledek. V šachu se ale dá využít faktu, že není nutné znát výsledek, abychom přišli na to, že jeden z hráčů má lepší pozici. Máme evaluační funkci, kterou můžeme použít. Následně při fázi backpropagation budeme průměrovat ohodnocení, která jsme dostali.

Nejlepší tah potom bude tah, který vede v průměru k nejlepšímu ohodnocení. Následně se dá testovat, do jaké hloubky chceme simulaci provádět a vliv změny této hloubky na výkon enginu.

7.2 Implementace

Jako první jsem se rozhodl implementovat nejjednodušší variantu MCTS, kde hledáme maximálně do hloubky 30. Fáze expanze tedy může dosáhnout maximálně hloubky 20. Nejpozději v této hloubce dojde na fázi simulace, která je prováděna do hloubky 10.

Fáze simulace je potom prováděna funkcí PlayAndRate, kterou provádíme, pokud je příznak expand false. Funkce PlayAndRate hraje náhodné tahy do hloubky, kterou stanovíme. Jakmile dosáhne dané hloubky, vrátí ohodnocení šachovnice.

Při procházení stromu potom kontrolujeme, zda uzel, na který jsme narazili, je nový. Jinými slovy je to uzel, který jsme ještě nezkoumali. Pokud ano, klíč tohoto uzlu bude nastaven na 0. V tomto případě jsme narazili na fázi expanze, kde přidáváme nový uzel do vyhledávacího stromu. Vygenerujeme tedy klíč a přípustné tahy a nastavíme příznak expand na false, aby v dalším rekurzivním volání došlo k simulaci.

Následně náhodně vybereme tah a zahrajeme ho. Pochopitelně musím pořád kontrolovat, zda je tah validní. Pokud náhodně vybraný tah není validní, vybereme jiný. Pokud 50 náhodně vybraných tahů není validních, považujeme pozici za mat.

Pokud ale najdeme validní tah, vytvoříme nový uzel pro index tahu a rekurzivně zavoláme MCTS. Z volání obdržíme ohodnocení, které nám vrátila fáze simulace. Následně spočítáme skóre tahu, abychom později mohli upravovat vybírání tahů. Přičteme výsledek k ohodnocení uzlu a inkrementujeme počet simulací.

Pokud jsme ale při fázi výběru uzlu narazili na již expandovaný uzel (uzel, který má klíč rozdílný od 0), vybereme náhodně tah, který chceme simulovat. Pokud tah není validní, nastavíme mu negativní skóre, abychom jej už neprocházeli.

Následně ověříme, zda tah vede k již existujícímu uzlu. Pokud ne, vytvoříme novou SearchMove strukturu a rekurzivně zavoláme MCTS.

Následně nastavíme hodnotu skóre a počtu vyhledávání jak u uzlu, tak u tahu. Hodnota u tahu bude použita k upravení funkce vybírání tahu.

Při vyhledávání voláme MCTS, dokud nám nevyprší čas. Jakmile vyprší čas, který byl stanovený na provedení tahu, projdeme kořen a vypočítáme nejlepší tah. Jinými slovy procházíme všechny expandované uzly a vybereme ten, který má nejlepší průměrné skóre.

```
1     node->key = simpleHash(bitboard);
2     (*bitboard).GenerateMoves(node->moveList);
3     expand = false;
4     //Pick the move to simulate (Expand or create new node)
5     char moveIndex = (std::rand() % (node->moveList->length));
6     node->moveList->moves[moveIndex].keyBeforeMove = node->key;
7     char tests = 0;
8     while (!(*bitboard).PlayMove(&node->moveList->moves[moveIndex]))
          {
9       (*bitboard).UndoMove();
10      //Punish the move
11      node->moveList->moves[moveIndex].score = -INFINITE;
12      moveIndex = (std::rand() % (node->moveList->length));
13      if (tests > 50) {
14        return -INFINITE - depthLeft;;
15      }
16      tests++;
17      node->moveList->moves[moveIndex].keyBeforeMove = node->key;
18    }
19    if (node->nodeList[moveIndex] == NULL) {
20      node->nodeList[moveIndex] = new SearchNode();
21    }
22    int temp = -MCTS(s, depthLeft - 1, !whiteOnMove, node->nodeList[
          moveIndex], bitboard, expand);
23    node->score += temp;
24    node->searches += 1;
25    node->moveList->moves[moveIndex].score += temp;
26    node->moveList->moves[moveIndex].searches += 1;
27    bitboard->UndoMove();
```

Zdrojový kód 28: Monte Carlo vyhledávání: nový uzel

```
1     char moveIndex = SelectMove(node->moveList);
2     if (moveIndex == -1) {
3       return -INFINITE - depthLeft;;
4     }
5     node->moveList->moves[moveIndex].keyBeforeMove = node->key;
6     char tests = 0;
7     while (!(*bitboard).PlayMove(&node->moveList->moves[moveIndex]))
          {
8       (*bitboard).UndoMove();
9       //Punish the move
10      node->moveList->moves[moveIndex].score = -INFINITE;
11      moveIndex = SelectMove(node->moveList);
12      if (moveIndex == -1) {
13        return -INFINITE - depthLeft;;
14      }
15      tests++;
16      node->moveList->moves[moveIndex].keyBeforeMove = node->key;
17    }
18    //Expand it
19    if (node->nodeList[moveIndex] == NULL) {
20      node->nodeList[moveIndex] = new SearchNode();
21    }
22
23    int tmp = -MCTS(s, depthLeft - 1, !whiteOnMove, node->nodeList[
          moveIndex], bitboard, expand);
24    node->searches += 1;
25    node->score += tmp;
26    node->moveList->moves[moveIndex].score += tmp;
27    node->moveList->moves[moveIndex].searches += 1;
28    bitboard->UndoMove();
```

Zdrojový kód 29: Monte Carlo vyhledávání: existující uzel

7.3 Další verze

Další verze bude velice podobná. Vše bude stejné jako v předchozí verzi, nicméně výběr tahů při selekci nebude probíhat náhodně. Naopak se budeme snažit najít rovnováhu mezi procházením nových tahů a zkoumáním nejlepších tahů.

Při výběru tahů ve fázi selection je tedy každý pátý tah vybrán jako tah s nejlepším skóre. Další pětina tahů je vybrána jako tah s nejlepším průměrným skóre. Průměrné skóre spočítáme jednoduše vydělením skóre tahu počtem vyhledávání tahu. Ve zbývajících třech pětinách tahů potom vybíráme tahy náhodně.

```
1     //pick the best score
2     char index = 0;
3     int bestYet = INT_MIN;
4     for (char i = 0; i < moves->length; i++) {
5       if (moves->moves[i].score > bestYet) {
6         bestYet = moves->moves[i].score;
7         index = i;
8       }
9     }
10    return index;
```

Zdrojový kód 30: Monte Carlo: výběr tahu

```
1     //pick the best score to searches ratio
2     char index = 0;
3     int bestYet = INT_MIN;
4     for (char i = 0; i < moves->length; i++) {
5       if (moves->moves[i].searches == 0) {
6         continue;
7       }
8       if ((moves->moves[i].score) / (moves->moves[i].searches) >
            bestYet) {
9         bestYet = (moves->moves[i].score) / (moves->moves[i].searches)
              ;
10        index = i;
11      }
12    }
13    return index;
```

Zdrojový kód 31: Monte Carlo: výběr tahu

8 Paralelní Monte Carlo vyhledávání

Při vyhledávání Monte Carlo metodou by nám mohlo pomoct paralelní vyhledávání. Jelikož procházení tahů a uzlů je z velké části náhodné, nezáleží tolik na pořadí. Navíc zde nenastává žádné ořezávání a nepoužíváme transpoziční tabulku.

To vypadá jako ideální prostředí pro možnost zlepšit engine. Velice jednoduše jsem tedy upravil kód vyhledávání tak, aby se dal provádět paralelně. Jediné místo, kde by mohl nastat problém, je při nastavování skóre a počtu vyhledávání. Přidal jsem tedy na toto místo zámek. Kód ve vyhledávání tedy vypadá následovně:

```
1     search_add.lock();
2     node->searches += 1;
3     node->score += tmp;
4     node->moveList->moves[moveIndex].score += tmp;
5     node->moveList->moves[moveIndex].searches += 1;
6     search_add.unlock();
```

Zdrojový kód 32: Monte Carlo paralelní synchronizace

Dále jsem vytvořil funkci parallelMCTS, na které budeme startovat jednotlivá vlákna. Funkce provádí neustále MCTS, dokud není nastaven příznak konce vyhledávání. Tento příznak nastavuje hlavní vlákno, jakmile vyprší čas.

```
1 void parallelMCTS(SearchData* s, int depthLeft, bool whiteOnMove,
     SearchNode *node, Bitboard* bitboard, bool expand, bool *end) {
2   while (!end) {
3     MCTS(s, depthLeft, whiteOnMove, node, bitboard, expand);
4   }
5 }
```

Zdrojový kód 33: Monte Carlo paralelní vyhledávání

Následně ve funkci, kde startujeme vyhledávání, provádíme následující kód, který se stará o spuštění vláken a následně provádí MCTS na hlavním vlákně v iteracích o 100 000 vyhledávání. Jakmile vyprší čas, nastavíme příznak end a ukončíme vyhledávání.

```
1   std::thread threads[NUM_OF_THREADS];
2   bool end = false;
3   for (int j = 0; j < NUM_OF_THREADS; j++) {
4    threads[j] = std::thread(parallelMCTS, sd, sd->maxDepth, white, &
        root, bForThread[j], true, &end);
5   }
6   do {
7    for (int i = 0; i < SEARCHES_BETWEEN_MOVES; i++) {
8     MCTS(sd, sd->maxDepth, white, &root, bitboard, true);
9    }
10   finish = std::chrono::high_resolution_clock::now();
11   microseconds = std::chrono::duration_cast<std::chrono::
        milliseconds>(finish - start);
12  } while (sd->timeOnMove > microseconds.count() * 3);
13  end = true;
```

Zdrojový kód 34: Monte Carlo paralelní vyhledávání

8.1 Verze enginu

Při paralelním MCTS se dá nastavit různý počet vláken a otestovat vliv na výkon enginu. Rozhodl jsem se tedy vytvořit několik variant enginu s různým počtem vláken. Rozhodl jsem se pro 4, 8, 16, 32 a 64 vláken.

9 Testování Monte Carlo vyhledávání

Při testování jsem zvolil formát 1minutových her, kde každý z hráčů má na partii jednu minutu. Dále, jelikož je hodně verzí na testování, jsem se rozhodl testovat enginy Monte Carlo samostatně proti předchozí verzi enginu. Každý engine hraje proti předchozí verzi 50 partií.

Bohužel při testování jsem přišel na to, že engine využívající Monte Carlo vyhledávání je mnohem horší než klasický engine využívající minimax. Engine využívající Monte Carlo vyhledávání hraje tahy až skoro náhodně a hra u všech verzí MCTS vedla k prohře již po relativně malém počtu tahů.

Je totiž velice jednoduché, aby jedno z vláken při konci hledání prošlo tah, který náhodou vede k skvělému ohodnocení. Problém ale je, že často při simulaci nejsou hrány dobré tahy. A výsledné ohodnocení je tedy velmi hrubý odhad. Počet pozic, které engine s MCTS projde během pár sekund, je přes 1 000 000, to ovšem není moc, když vezmeme v potaz fakt, že alpha beta engine projde asi trojnásobek pozic a navíc má přesné ohodnocení a ne hrubý odhad. Při přidání vláken do MCTS engine projde více pozic, nicméně pořád je to méně než klasický engine využívající alpha beta vyhledávání a i verze s paralelním MCTS prohraje během pár tahů.

Výsledek je tedy jasný, pro šachový engine, který nevyužívá neuronové sítě, je lepší použít minimax algoritmus s různými funkcemi pro zrychlení prohledávání.

10 Evaluační funkce

Evaluační funkce je jedna z nejdůležitějších částí enginu. Ve vyhledávání a procházení stromu je pouze omezené množství heuristik, které se dají implementovat pro zlepšení výkonu enginu. U evaluační funkce je ale možností více. I velice malá změna v evaluační funkci vede k velkému rozdílu ve výkonu enginu. Čtenář se o tom bude moct přesvědčit v další kapitole.

10.1 Evaluace s bitovými šachovnicemi

V bakalářské práci jsem vytvořil engine, který používal reprezentaci šachovnice pomocí pole. Při ohodnocení šachovnice se tedy logicky pracovalo s polem. To je ale neefektivní, jelikož v nové verzi enginu používáme bitové šachovnice. Můžeme při ohodnocování pozice využít bitové operace. Ty jsou mnohem rychlejší na provedení.

Díky rychlosti operací máme možnost zařadit do ohodnocující funkce více faktorů. Například můžeme mnohem přesněji ohodnotit bezpečnost krále. V bakalářské práci by se nevyplatilo přesnější ohodnocení, protože čas strávený ohodnocením by byl moc velký.

10.2 Implementace

Implementace evaluační funkce je, jak jsem zmínil, postavená na bitových operacích. Ty byly v textu popsány v předchozích kapitolách. Nebudu je tedy vysvětlovat znovu.

Evaluační funkce mění způsob ohodnocení na základě toho, v jaké části partie se nacházíme. Je třeba odlišit ohodnocení v koncovce od ohodnocení ve střední hře. V koncovce bere ohodnocující funkce v potaz následující faktory pro ohodnocení šachovnice:

- materiál na šachovnici,
- postavení figur,
- důraz na pozici krále,
- mobilitu figur,
- dvojici střelců,
- úroveň pěšců,
- pěšce na stejném sloupci,
- izolované pěšce,
- věž na volném sloupci.

Engine rozhoduje, zda se hra nachází v koncovce podle počtu figur na šachovnici. Pokud je hodnota materiálu bílého menší než 1800, engine ohodnocuje pozici jako pozici v koncové hře, pokud je vyšší, ohodnocuje engine pozici jako pozici ve střední hře. Čtenáře by mohlo zajímat, proč se používá pouze materiál bílého hráče a materiál černého hráče se při rozhodování ignoruje. Ve velkém množství partií je materiál na šachovnici přibližně vyrovnaný, kontrola figur černého hráče by byla ve velkém množství pozic redundantní. Navíc by kontrola černého mohla přinést případy, kdy například černý hráč bude mít materiál v hodnotě 1900 a bílý hráč materiál v hodnotě 1700. Muselo by se tedy následně rozhodnout, kterou z hodnot ignorovat. Ve střední hře a zahájení bere ohodnocující funkce v potaz následující faktory:

- materiál na šachovnici,
- postavení figur,
- kontrolu středu šachovnice,
- mobilitu figur,
- dvojici střelců,
- pěšce před králem,
- provedení rošády,
- krále na prázdném sloupci,
- pozice krále,
- dámu na volném sloupci,
- úroveň pěšců,
- pěšce na stejném sloupci,
- izolované pěšce,
- věž na volném sloupci.

```
1     //KING POSSITION
2     U64 t = b->whitePiecesList[5];
3     char whiteKingSq = POP(&t);
4     rating += KingE[whiteKingSq];
5     t = b->blackPiecesList[5];
6     char blackKingSq = POP(&t);
7     rating -= KingE[mirrorSquare[blackKingSq]];
8
9     //MOBILITY RATE
10    rating += (b->countMoves(true) - b->countMoves(false)) * 3;
11
12    // Double bishop
13    rating += COUNT(b->whitePiecesList[2]) * 5;
14    rating -= COUNT(b->blackPiecesList[2]) * 5;
15
16    //PASSED PAWNS
17    U64 temp = b->whitePiecesList[0];
18    while (temp) {
19      char sq = POP(&temp);
20      if (!(passed_pawns_offsets_white[sq] & b->blackPiecesList[0]))
            {
21        rating += PawnPassed[rank_of_square[sq]];
22      }
23    }
24    temp = b->blackPiecesList[0];
25    while (temp) {
26      char sq = POP(&temp);
27      if (!(passed_pawns_offsets_black[sq] & b->whitePiecesList[0]))
            {
28        rating -= PawnPassedBlack[rank_of_square[sq]];
29      }
30    }
```

Zdrojový kód 35: Ukázka kódu evaluační funkce

11 Ladění enginu

Ve výše zmíněných kapitolách jsem se pokusil implementovat určité heuristiky pro zlepšení výkonu šachového enginu. Některé byly velice úspěšné, jiné měly minimální nebo až záporný efekt na hru. V této kapitole se budu zabývat vyladěním enginu tak, aby hrál co možná nejlépe. Úprava bude probíhat především v ohodnocující funkci.

Při testování jednotlivých úprav jsem nechal hrát starší verzi enginu s novou verzí.

11.1 Cennější střelec a jezdec

První verze enginu testuje, zda je lepší zvýhodnit střelce a jezdce při ohodnocování šachovnice. Někteří autoři to preferují, jiní říkají, že to nedává smysl. Já jsem se osobně stavil na stranu odpůrců. Už odmala jsem se učil, že střelec a jezdec je roven třem pěšcům. O to více jsem byl překvapen výsledky při testování. Nová verze, používající pro ohodnocení střelce a jezdce hodnotu 350, měla kladnou bilanci se starší verzí, která používala hodnotu 300.

1 minutové partie	Výhry	Prohry	Remízy
Starší verze	32	38	20
Nová verze	38	32	20

5 minutové partie	Výhry	Prohry	Remízy
Starší verze	7	17	13
Nová verze	17	7	13

Rozdíl sice není nijak významný u minutových her, nicméně u pětiminutových her je nová verze enginu lepší. Rozhodl jsem se tedy v enginu použít hodnotu 350 pro ohodnocení střelce a jezdce.

11.2 Cennější rošáda

Bezpečnost krále je jedna z nejdůležitějších věcí ve hře šachu. V další kapitole se čtenář dozví o všech faktorech, které jsou brány v potaz pro ohodnocení bezpečnosti krále. Pořád jsme ale kladli poměrně velkou důležitost na to, zda je provedena rošáda. Přidávali jsme 50 bodů jako bonus za provedenou rošádu.

Jelikož je ale ohodnocení bezpečnosti krále mnohem přesnější, můžeme zkusit snížit tento bonus za rošádu na 20 a vyzkoušet, která verze enginu je lepší.

1 minutové partie	Výhry	Prohry	Remízy
50bonus	23	19	19
20bonus	19	23	19

5 minutové partie	Výhry	Prohry	Remízy
50bonus	7	2	13
20bonus	2	7	13

Z výsledků je zajímavé, že i když se nejedná o drastickou změnu v enginu, jedná se o pouhé upravení konstanty, dopad na výkon enginu existuje. Můžeme si všimnout, že verze používající 50 bodů jako bonus má kladnou bilanci. Rozdíl je znát především u pětiminutových partií. Rozhodl jsem se ji tedy využít v enginu.

11.3 Zjednodušení ohodnocení v koncovce

Ohodnocující funkce musí být co nejrychlejší, ale také přesné. Je potřeba mít rovnováhu při tvorbě této funkce. Při koncovkách jsou pozice typicky jednodušší na ohodnocení, protože není tolik figur na šachovnici. Ve starší verzi ohodnocovací funkce ale pořád používáme bonus za mobilitu figur, který není jednoduchý na spočítání. V koncovce ale není až tak důležité mít mobilní figury, jelikož typicky všechny figury mají dostatek prostoru. Zkusil jsem tedy odstranit tento faktor ohodnocení.

Dále jsem odstranil faktor ohodnocení věží na prázdných sloupcích. Opět stejné důvody, v koncovce už je typicky málo figur, takže je velká pravděpodobnost, že věž bude na nějakém z volných sloupců.

Starší verze tedy používá výše zmíněné faktory, nová verze je nepoužívá.

1 minutové partie	Výhry	Prohry	Remízy
Starší verze	15	20	31
Nová verze	20	15	31

5 minutové partie	Výhry	Prohry	Remízy
Starší verze	10	6	14
Nová verze	6	10	14

Na výsledcích může čtenář vidět, že má nová verze kladnou bilanci u minutových partií, ale u delších partií je přesnější ohodnocení pozice výhodnější. Rozhodl jsem se tedy použít starší verzi.

11.4 Bonus za pozici figur

Bonus za pozici figur je jeden z nejdůležitějších faktorů při ohodnocování šachovnice. Dává enginu ponětí o tom, kam má postavit figury. Bonus je počítán pomocí polí o 64 elementech. Pro pěšce, střelce, jezdce a věž máme takové pole. Pro dámu a krále nemáme, jelikož jejich pozice na šachovnici je hodnocena jiným způsobem.

Při kontrole těchto konstant jsem si všiml drobností, které by enginu mohly pomoci.

Konstanty jsem upravil následovně:

- Pěšec: Přidání bonusu, pokud pěšec zůstává na F2 nebo F4 (z pohledu bílého) a přidání malého bonusu při posunutí pěšce na C3. Oba tyto faktory slouží především pro ochranu krále, mít pěšce na F3 ve většině případů není úplně vhodné, když uděláme rošádu na královském křídle. Na druhou stranu pěšec na C3 může pomoci s ovládnutím středu šachovnice.

- Jezdec: Odstranění bonusů za pozici na protihráčově polovině a penalizace jezdců na kraji šachovnice.

- Střelec: Přidání bonusu pro střelce na hlavní diagonále a zvýšení bonusu pro střelce na G2 a B2, jelikož tato pole jsou považována za jedna z nejlepších pro střelce.

- Věž: Přidání malých záporných ohodnocení pro věž, která se nachází na výchozím poli. Jedná se opravdu o malé záporné ohodnocení 5 bodů. Myšlenka je, že pokud engine nemá moc dobrých tahů, může posunout věž na jeden ze sloupců uprostřed, což je typicky lepší pozice.

Změny nejsou nějakým způsobem závratné. Jedná se spíše o doladění. Zajímalo mě, jak velký budou mít změny vliv na výkon enginu.

1 minutové partie	Výhry	Prohry	Remízy
Starší verze	26	14	23
Nová verze	14	26	23

5 minutové partie	Výhry	Prohry	Remízy
Starší verze	11	11	8
Nová verze	11	11	8

Výsledek testování ovšem ukázal, že změna bonusových polí z nějakého důvodu vedla k horší hře u partií s kratším časovým limitem. Rozhodl jsem se tedy použít starší verzi.

11.5 Zvýraznění důležitosti ovládání středu

V šachu je známé nepsané pravidlo: hráč, který ovládá střed, ovládá partii. Evaluační funkce již zohledňuje, který z hráčů ovládá střed. Zajímalo mě ale jaký vliv by mělo zvýraznění tohoto faktoru.

Ve starší verzi bereme čtverec obsahující 16 polí uprostřed šachovnice a za každou figuru uvnitř přidáváme 10 bodů. V nové verzi přidáváme 10 bodů pěšcům a 5 bodů ostatním figurám v tom samém čtverci, ale navíc máme druhý čtverec obsahující pouze 4 pole ve středu šachovnice. U tohoto čtverce dáváme bonus dalších 20 bodů pro pěšce a 10 pro ostatní figury.

1 minutové partie	Výhry	Prohry	Remízy
Starší verze	29	17	26
Nová verze	17	29	26

5 minutové partie	Výhry	Prohry	Remízy
Starší verze	20	20	18
Nová verze	20	20	18

Dle výsledků je vidět, že změna ohodnocení šachovnice neměla požadovaný efekt. Rozhodl jsem se tedy nezvýhodňovat důležitost středu šachovnice.

11.6 Útočící a bránící figury

Momentálně používáme pro ohodnocení šachovnice pouze mobilitu figur. Jinými slovy počítáme možné tahy a násobíme je konstantou. Nebereme už ale v potaz podporu figur a napadení figur soupeře.

V reálné hře to ale tvoří velice důležitý faktor ohodnocení pozice. Především když máme nějaké slabé pole, které je třeba bránit nebo z druhého pohledu, na které můžeme zaútočit.

Rozhodl jsem se tedy do ohodnocující funkce přidat funkcionalitu pro ohodnocení útočících a bránících tahů, kde využíváme metodu ze třídy Bitboard pro počítání přípustných tahů. Ta následně pro každou figuru spočítá, kolik je možných tahů, útočících tahů a kolik vlastních figur figura podporuje. Kód 37 pro počítání takových tahů pro střelce. Kód je jen pro jeden směr pohybu střelce. Ostatní směry jsou implementovány obdobně, jen používají jiný ofset. Ostatní figury jsou implementovány obdobně.

Při testování byla nová verze enginu značně horší. Je možné, že overhead, který s sebou počítání útočících a bránících tahů nese, převážil benefity. Logicky jsem se tedy rozhodl tuto funkcionalitu do enginu nepřidávat.

```
1     //MOBILITY RATE
2     int wAttacks = 0, wMoves = 0, wSupport = 0, bAttacks = 0, bMoves
          = 0, bSupport = 0;
3     b->countMoves(true, &wMoves, &wAttacks, &wSupport);
4     b->countMoves(false, &bMoves, &bAttacks, &bSupport);
5     //MOBILITY RATE
6     rating += (wMoves - bMoves) * 3;
7     rating += (wAttacks - bAttacks) * 5;
8     rating += (wSupport - bSupport) * 2;
```

Zdrojový kód 36: Implementace útočící a bránící tahy

```
1   //Inicializace
2   U64 temp = bishops;
3   U64 t;
4   U64 cMyPieces = ~myPieces;
5   U64 cOpponentPieces = ~opponentPieces;
6   // LEFT UP
7   while (temp) {
8     temp = (((temp & NOT_H_FILE) & (~ROW_8)) << LEFT_UP);
9     t = temp & opponentPieces;
10    *attacks = *attacks + COUNT(t);
11    t = temp & myPieces;
12    *support = *support + COUNT(t);
13    temp &= cOpponentPieces;
14    temp &= cMyPieces;
15    *moves = *moves + COUNT(temp);
16  }
```

Zdrojový kód 37: Implementace počítání tahů

1 minutové hry	Výhry	Prohry	Remízy
Starší verze	38	9	10
Nová verze	9	38	10

5 minutové hry	Výhry	Prohry	Remízy
Starší verze	10	2	3
Nová verze	2	10	3

11.7 Věž za pěšcem v koncovce

V koncovce je důležité mít pěšce co nejdál a mít je nějakým způsobem podpořeny. Klasicky se dají podpořit pomocí věže, která se nachází na stejném sloupci. V

předchozí evaluační funkci nebyla přidána žádná taková funkcionalita.

V nové verzi tedy kontrolujeme, zda se věž nachází za pěšcem a následně přidáváme bonus, který se odvíjí od řádky, na které se pěšec nachází.

1 minutové partie	Výhry	Prohry	Remízy
Starší verze	16	1	6
Nová verze	1	16	6

5 minutové partie	Výhry	Prohry	Remízy
Starší verze	8	0	0
Nová verze	0	8	0

Rozdíl mezi těmito verzemi je drastický. Pravděpodobně opět nastal případ, kdy overhead, který funkcionalita přináší, převažuje benefit přesnějšího ohodnocení.

11.8 Podpora pěšců

Podpora pěšců je další důležitý faktor. Pokud je totiž pěšec osamostatněný, stává se z něj typicky slabé místo vhodné pro útok.

Rozhodl jsem se tedy přidat funkci ohodnocující podporu pěšců. Nejdůležitější je mít pěšce podpořeny jinými pěšci. To je ale díky binárním operacím velice jednoduché na spočítání.

```
1    temp = ((b->whitePiecesList[0] & NOT_H_FILE) << LEFT_UP) | ((b->
        whitePiecesList[0] & NOT_A_FILE) << RIGHT_UP);
2    temp = b->whitePiecesList[0] & (~temp);
3    rating -= COUNT(temp) * unsupportedPawnEnd;
4
5    temp = ((b->blackPiecesList[0] & NOT_A_FILE) >> LEFT_UP) | ((b->
        blackPiecesList[0] & NOT_H_FILE) >> RIGHT_UP);
6    temp = b->blackPiecesList[0] & (~temp);
7    rating += COUNT(temp) * unsupportedPawnEnd;
```

Zdrojový kód 38: Implementace podpora pěšců

Nejdříve si do proměnné temp uložíme body na šachovnici, kde pěšci útočí (brání z našeho pohledu). Následně si jen spočítáme počet pěšců, kteří nejsou žádným jiným pěšcem podpořeni. Následně provedeme penalizaci, za každého takového pěšce v koncovce odečítáme 10 bodů.

1 minutové partie	Výhry	Prohry	Remízy
Starší verze	15	4	3
Nová verze	4	15	3

5 minutové partie	Výhry	Prohry	Remízy
Starší verze	9	6	12
Nová verze	6	9	12

Implementace této funkce měla zápornou bilanci oproti předchozí verzi enginu. Rozhodl jsem se tedy tuto funkci nepřidávat.

11.9 Zvýšení bezpečnosti krále

Při sledování hry enginu jsem si všiml, že engine často prohrává, protože se dostane pod tlak. Protihráč vytváří tlak na krále, který vyústí v to, že engine musí obětovat materiál, aby nedostal mat.

Rozhodl jsem se tedy zvýšit důležitost bezpečnosti krále při ohodnocení pozice. Pokud je král na sloupci, na kterém se nenachází protivníkův pěšec, dostane hráč penalizaci 50 bodů. Pokud se navíc na takovém sloupci nachází protivníkova věž, dostává penalizace dalších 15 bodů. V předchozí verzi byla penalizace jen 20 bodů.

Engine navíc dává bonus za to, že se v okolí krále nachází libovolná figura. V předchozí verzi se dával bonus pouze za pěšce, a to pouhých 5 bodů. V nové verzi dostává engine bonus 6 bodů za libovolnou figuru a 12 bodů za pěšce před králem.

Dále jsem odebral ohodnocení úrovně pěšců ve střední hře, jelikož to není důležitý faktor a navíc je již bonus udělen při ohodnocení pozice pěšce.

1 minutová partie	Výhry	Prohry	Remízy
Starší verze	13	16	7
Nová verze	16	13	7

5 minutové partie	Výhry	Prohry	Remízy
Starší verze	11	3	15
Nová verze	3	11	15

Nová verze enginu má kladnou bilanci oproti starší verzi u minutových partií. Nicméně bilance u pětiminutových partií je záporná. Rozhodl jsem se tedy nepřidávat tuto funkcionalitu.

11.10 Kombinace více faktorů

V předchozích verzích jsem testoval pouze jedno vylepšení izolovaně. V této verzi enginu jsem se rozhodl kombinovat více faktorů. První je zvýšení hodnoty střelce a jezdce na 350 bodů. Druhý je zjednodušení ohodnocování šachovnice v koncovce. Nově v ohodnocující funkci nepoužíváme bonusová pole za pozici figur v koncovce. Bonusová pole používáme jen v zahájení a střední hře. Navíc jsem se rozhodl upřesnit výpočet dvojice střelců. V nové verzi enginu kontroluji, zda se na šachovnici nachází alespoň dva střelci. Pokud ano, hráč dostává bonus 30 bodů. Ve starší verzi enginu tento bonus funguje bez kontroly, zda se na šachovnici opravdu nacházejí 2 střelci. Jednoduše přidává bonus roven pětinásobku počtu střelců na šachovnici.

1 minutové partie	Výhry	Prohry	Remízy
Starší verze	21	16	9
Nová verze	16	21	9

5 minutové partie	Výhry	Prohry	Remízy
Starší verze	16	28	14
Nová verze	28	16	14

Na výsledcích je vidět, že tato kombinace faktorů, enginu pomohla rychleji ohodnotit pozici bez ztráty na přesnosti ohodnocení. V koncovce je totiž postavení figur méně důležité než v zahájení a střední hře. Zajímavé je, že bilance je záporná u minutových partií, zatímco u pětiminutových partií je kladná.

11.11 Kombinace více faktorů 2

Počítání polí, na které hráč útočí, je u výkonných enginů běžná věc. V této práci jsem ale již přišel na to, že pokud přidám jen tuto funkcionalitu bez žádné další změny, pak má engine horší výkonnost. V této verzi jsem se tedy pokusil kombinovat počítání útočících a bránících tahů s odstraněním bonusových polí. Bonusová pole jsou dobrá především v zahájení, kde enginu pomohou postavit správně figury. V ostatních částech hry už je ale mobilita figur důležitější. Aby engine dokázal přesněji ohodnotit, jaká pozice je dobrá, rozhodl jsem se zvýšit bonusy za mobilitu.

1 minutové partie	Výhry	Prohry	Remízy
Starší verze	24	3	3
Nová verze	3	24	3

```
1     int wAttacks = 0, wMoves = 0, wSupport = 0, bAttacks = 0, bMoves
          = 0, bSupport = 0;
2     b->countMoves(true, &wMoves, &wAttacks, &wSupport);
3     b->countMoves(false, &bMoves, &bAttacks, &bSupport);
4     //MOBILITY RATE
5     rating += (wMoves - bMoves) * 5;
6     rating += (wAttacks - bAttacks) * 8;
7     rating += (wSupport - bSupport) * 4;
```

Zdrojový kód 39: Implementace útočící a bránící tahy

5 minutové partie	Výhry	Prohry	Remízy
Starší verze	10	0	1
Nová verze	0	10	1

Výsledek testování ukázal, že tato kombinace faktorů nefunguje dobře. Použití bonusových polí je poměrně jednoduché na výpočet a především v zahájení přinutí engine správně postavit figury.

12 AlphaZero šachový engine

Před pár lety přišla společnost Alphabet (mateřská společnost Google) s šachovým enginem, který zcela změnil přístup k tvorbě šachových enginů. Šachový engine AlphaZero používá neuronové sítě na ohodnocení šachových pozic. Následně místo minimax vyhledávání s alphabeta ořezáváním používá MCTS (Monte Carlo Tree Search) algoritmus, který jsme testovali v kapitole výše. Při testování verze enginu používající MCTS prohrála každou partii, nicméně u AlphaZero funguje velice dobře pro vyrovnání chyb z ohodnocení šachovnice. Když byl tento engine porovnán s momentálně nejlepším enginem na světě, ani jednou neprohrál. Nicméně tento test je dle některých expertů zavádějící, jelikož AlphaZero běžel na lepším hardwaru, který byl speciálně upravený pro jeho neuronovou síť.

12.1 Princip AlphaZero

Společnost Alphabet se už dlouho zabývá tvorbou a zkoumáním umělé inteligence. Motivací pro vytvoření AlphaZero bylo vytvořit program, který dokáže překonat člověka nejen ve hře šachu, ale také v ostatních hrách jako například go. To je úplně jiný přístup. Ostatní enginy se soustředí pouze na jednu hru.

Alphabetu se povedlo vytvořit obecný algoritmus, který dokáže hrát více her na úrovni nejlepších hráčů na světě. AlphaZero nepoužívá klasický brute force algoritmus jako například Stockfish. AlphaZero v průměru prošlo pouze 63 tisíc pozic za sekundu, zatímco Stockfish procházel přes 58 milionů pozic za sekundu.

AlphaZero používá následující technologie:

- Convolutional neural networks - pro automatickou analýzu šachovnice a extrakci hlavních charakteristik pozice.
- Deep reinforcement learning - použité pro trénování neuronových sítí.
- Neural network residual layers - umožňuje přidání vrstev do konvolučních neuronových sítí.
- Batch normalization - pro normalizaci hodnot na vstupu neuronové sítě.

12.2 Architektura AlphaZero

AlphaZero používá 2 neuronové sítě. První se nazývá Policy Network. Ta slouží k rozhodnutí, jaký bude další tah. Tato síť přijímá šachovnici jako vstup a výsledkem je velmi přesné ohodnocení jednotlivých tahů v dané pozici. Pro vytrénování této sítě AlphaZero používá pouze hraní proti sobě sama. Policy Network se používá především pro určení, jakým tahem se má MCTS vyhledávání zabývat. Druhá síť se nazývá Value Network. Ta slouží k ohodnocení pozice. Vstup je tedy šachovnice a výstup je hodnota reprezentující, který z hráčů v dané pozici pravděpodobně vyhraje.

Policy Network slouží především k redukci branching factoru u MCTS tím, že přesně odhadne nejlepší tahy v dané pozici. Value Network následně slouží k redukci hloubky vyhledávání. V libovolném bodu vyhledávání se můžeme rozhodnout, jestli pozici ohodnotíme. Pokud pozice často vede k prohře, pravděpodobně není nutné procházet pozici dál.

13 Neurnová síť pro ohodnocení pozice

13.1 Motivace

Navzdory tomu, co si většina lidí myslí, rozdíl mezi nejlepšími hráči šachu není v tom, kolik tahů si dokáží představit dopředu. Rozdíl je spíše v tom, jak hráč dokáže odhadnout, který tah je v pozici nejlepší. Pro profesionálního hráče šachu je typicky důležitější přijít s nápaditým tahem než počet tahů, které vidí dopředu.

Engine, který je vytvořený v této práci, momentálně dává větší důraz na hloubku hledání, ne na to najít nápaditý tah v pozici. Engine se snaží projít co nejvíce tahů a zmenšit při tom branching factor. Na tomto principu funguje ale drtivé množství šachových enginů. I ten, který je momentálně nejvýkonnější na světě. Tyto enginy implementují heuristiky na seřazení tahů. Tyto heuristiky jsou ovšem často dosti primitivní a nemohou se rovnat schopnostem velmistra.

13.2 Architektura a vstup

Každý hráč může mít na šachovnici maximálně 6 figur. Vstup je tedy dlouhý 12 * 64 bitů, neboli 768 bitů. Navíc musíme přidat informaci o tom, jaký hráč je na tahu. Dle veřejně dostupných vědeckých článků [**7**] vyšla nejlépe možnost sestrojit neuronovou síť, která bude mít 3 skryté vrstvy, kde každá vrstva bude mít 2048 perceptronů. Tato neuronová síť bude používat elu funkci pro výpočet zahoření.

Jak si čtenář dokáže představit, učení takové neuronové sítě zabere obrovské množství času a výpočetní síly. V celé neuronové síti je asi 10 milionů neznámých parametrů.

13.3 Výsledek

S ohledem na časovou náročnost a na obrovské množství možných šachových pozic je těžké vytrénovat neuronou síť tak, aby s přesností a jistotou dokázala ohodnotit danou pozici. Proto se také v šachových enginech, které používají neuronové sítě na ohodnocení šachovnice, často používá MCTS metoda na procházení stavového prostoru namísto MiniMax algoritmu s AlphaBeta ořezáváním, který se typicky používá u enginů, které mají experty sestavenou ohodnocující funkci.

Neuronová síť používaná v AlphaZero se učila 14 dní na superpočítači, který google přímo sestrojil pro učení neuronových sítí. Následně pak engine běží na procesorech, které jsou opět přizpůsobeny výpočtu neuronové sítě. Z důvodu chybějící výpočetní kapacity jsem se tedy rozhodl tuto funkcionalitu neimplementovat.

14 Výsledný engine

14.1 Testování výsledného enginu

Testování výsledného enginu jsem se rozhodl provádět stejným způsobem jako ostatní testování v této práci. Nechám enginy hrát proti sobě v minutových partiích, kde má každý z enginů jednu minutu na celou partii.

Pro testování jsem vybral engine Cinnamon, který má elo 2112 bodů. Hráč s takovým elem často dosahuje na titul kandidát mistra (FM). Byl jsem příjemně překvapen výsledkem u minutových partií, kde měl engine vytvořený jako diplomová práce kladnou bilanci. Dle výsledků minutových partií by se tedy dalo odhadovat elo enginu někde pod hranicí 2200 bodů.

U pětiminutových partií má engine zápornou bilanci. Je velmi těžké odhalit důvod. S největší pravděpodobností bude problém s rozložením času na jednotlivé tahy. Připadá mi, že engine tráví příliš času na výpočet tahů ve střední hře a následně má málo času v koncovce. Často jsem zaznamenal partii, kdy šel engine do koncovky v lepší pozici, ale s menším množstvím času a následně partii prohrál. U půhodinových partií měl engine opět negativní bilanci.

Minutové partie	Výhry	Prohry	Remízy
Diplomová práce	11	8	12
Cinnamon	8	11	12

Pětiminutové partie	Výhry	Prohry	Remízy
Diplomová práce	9	13	5
Cinnamon	13	9	5

Půlhodinové partie	Výhry	Prohry	Remízy
Diplomová práce	4	5	3
Cinnamon	5	4	3

Dále jsem se rozhodl engine otestovat proti enginu Buzz, který má Elo 2235 bodů. Engine vytvořený jako diplomová práce měl zápornou bilanci u všech typů partií. Problém je pravděpodobně u ohodnocení šachovnice, jelikož engine vytvořený jako diplomová práce v průměru procházel tahy do větší hloubky než protivník.

Minutové partie	Výhry	Prohry	Remízy
Diplomová práce	3	13	6
Buzz	13	3	6

Pětiminutové partie	Výhry	Prohry	Remízy
Diplomová práce	1	9	2
Buzz	9	1	2

Půlhodinové partie	Výhry	Prohry	Remízy
Diplomová práce	2	4	3
Buzz	4	2	3

Následně mě zajímalo, zda má engine výkon na to, aby porazil alespoň v jedné partii velmistra. Šachový velmistr má typicky Elo nad 2500 bodů. Pro testování jsem vybral engine Asymptote s elem 2505. Pochopitelně Asymptote měl výrazně kladnou bilanci. Nicméně engine vytvořený jako diplomová práce dokázal v jedné z minutových her vyhrát. U pětiminutových i půlhodinových partií prohrál engine všechny partie.

Minutové partie	Výhry	Prohry	Remízy
Diplomová práce	1	15	3
Asymptote	15	1	3

Pětiminutové partie	Výhry	Prohry	Remízy
Diplomová práce	0	12	0
Asymptote	12	0	0

Půlhodinové partie	Výhry	Prohry	Remízy
Diplomová práce	0	7	0
Asymptote	7	0	0

V případě, že si čtenář chce sám otestovat engine, může využít program Arena pro uspořádání turnaje mezi enginy. Přiložený soubor readme.txt obsahuje podrobný návod.

14.2 Porovnání se Stockfish

Engine vytvořený jak diplomová práce stále nedosahuje výkonu nejvýkonnějších enginů na světě. Nabízí se tedy otázka: proč tomu tak je?

Jelikož nejvýkonnější engine na světě, Stockfish, je open source, můžeme se podívat, jaký je rozdíl mezi enginy. Na první pohled si můžeme všimnout mnohem vyšší optimalizace kódu. Je uváděno, že Stockfish dokáže zpracovat až 70 milionů pozic za sekundu na moderním výkonném počítači. Když jsem ale stockfish testoval na mém počítači, toto číslo bylo mnohem nižší (přibližně 400 tisíc pozic). Engine vytvořený jako diplomová práce dokáže projít přibližně 800 tisíc pozic za sekundu. Na druhou stranu Stockfish dokáže provést hlubší vyhledávání než engine vytvořený jako diplomová práce. To je pravděpodobně jeden z největších důvodů, proč má Stockfish vyšší výkon.

Proč se dokáže dostat do vyšší hloubky, to se dá těžko odhadnout. Pravděpodobně to bude kombinací více faktorů. Stockfish dokáže lépe odhadnout, jaký tah procházet. Tráví tedy více času na rozhodnutí, který tah provést. Díky tomu projde Stockfish méně pozic za sekundu, ale na druhou stranu se dostane do větší hloubky.

Nepochybně tedy jeden z rozdílů je schopnost enginů odhadnout, který tah procházet dále a které tahy ořezat. V celkovém důsledku je nejdůležitější hloubka, do které se engine při vyhledávání dostane.

Zdrojový kód enginu Stockfish se dá jen velmi špatně číst, kód je optimalizovaný na rychlost a to pochopitelně ubírá na čitelnosti kódu. Stockfish používá principiálně velmi podobné vyhledávání. Rozdíl je pravděpodobně ve způsobu, jakým dokáže Stockfish provést toto vyhledávání. Například v enginu vytvořeném jako diplomová práce rozhodujeme, jaký tah projít pouze na základě brané figury. Stockfish naopak provádí vnitřní iterativní zlepšování, kdy prohledá v pozici tahy do snížené hloubky a následně na základě výsledků seřadí tahy.

Dále engine implementuje Futility ořezávání, které odstraní tahy, jež nemají možnost zvýšit hodnotu alpha ve vyhledávání. Stockfish tedy odhadne jaká bude hodnota tahu a přičte konstantu, která reprezentuje maximální chybu. Pokud je hodnota menší než alpha, tah je ořezán. Pro tuto funkcionalitu je třeba efektivně a přesně ohodnotit tah před ořezáním.

Zjednodušeně řečeno, Stockfish dělá lepší práci v rozhodování, které uzly procházet a které ne. Implementuje pro to různé heuristiky do vyhledávání, které v enginu, jenž jsem tvořil jako diplomovou práci, nejsou. Jinak je Stockfish principem vyhledávání velmi podobný enginu, který byl vytvořen jako diplomová práce.

Jednoznačně jedna ze slabostí enginu vytvořeného jako diplomová práce je slabší ohodnocovací funkce. Jak si může čtenář všimnout v kapitole 13, velmi malé změny v ohodnocovací funkci mají obrovský dopad na výkon enginu. Ohodnocovací funkce, kterou Stockfish využívá, je vyladěna šachovými experty, kteří mají bezpochyby mnohem vyšší znalost hry než já. Navíc je ohodnocovací funkce enginu Stockfish mnohem více optimalizovaná na rychlost.

Poslední velký rozdíl mezi enginem implementovaným jako diplomová práce

a nejvýkonnějším enginem na světě je schopnost nakládat efektivně s časem. Stockfish má velmi důmyslný systém, kde dokáže přesně odhadnout, v jaké pozici by měl trávit nejvíce času. Pro porovnání: engine vytvořený jako diplomová práce vydělí zbývající čas konstantou a rozlišuje pouze 2 fáze partie.

Závěr

Cílem práce bylo nastudování, implementace a otestování pokročilých metod tvorby šachového enginu. Práce pokračovala vylepšením již existujícího enginu, který jsem implementoval jako bakalářskou práci. Tento engine měl původně Elo okolo 1700 bodů. V diplomové práci jsem dokázal zlepšit výkonnost enginu. Odhadované Elo je někde pod úrovní 2200 bodů. Pro porovnání šachový velmistr má Elo nad 2500 bodů. Při testování a hře jednominutových partií dokonce dokázal engine porazit protivníka s elem přesahujícím 2500 bodů.

Práce je napsána tak, aby čtenáře seznámila s výsledky implementace pokročilých metod konstrukce šachového enginu. Po přečtení by měl být čtenář schopen implementovat engine podobné výkonnosti.

Conclusions

The goal of this thesis was to study, implement and test advanced methods of chess engine construction. This thesis was build on top of the work I did in my bachElor's thesis. The engine build during my bachElor thesis had about 1700 Elo points. In this thesis I managed to improve the performance of the engine. The resulting engine has Elo just under the 2200 mark. For comparison, the chess grand master has Elo above 2500 points. During the testing phase, the engine managed to beat another engine with Elo above 2500 points.

This thesis is written in a way so that the reader can understand the implementation and impact of different methods on the performance of chess engine. After reading this thesis, the reader should be able to create his own chess engine.

A Obsah přiloženého CD/DVD

bin/
: ChessEngine spustitelné přímo z CD/DVD.

doc/
: Text práce ve formátu PDF, vytvořený s použitím závazného stylu KI PřF UP v Olomouci pro závěrečné práce, včetně všech příloh, a všechny soubory potřebné pro bezproblémové vygenerování PDF dokumentu textu (v ZIP archivu), tj. zdrojový text textu, vložené obrázky, apod.

src/
: Kompletní zdrojové texty programu ChessEngine se všemi potřebnými (příp. převzatými) zdrojovými texty, knihovnami a dalšími soubory potřebnými pro bezproblémové vytvoření spustitelných verzí programu / adresářové struktury pro zkopírování na webový server.

readme.txt
: Instrukce pro instalaci a spuštění programu ChessEngine, včetně všech požadavků pro jeho bezproblémový provoz.

Bibliografie

[1] IJACSA, Ahmed A. Elnaggar, Mostafa Abdel Aziem, Mahmoud Gadallah, Hesham El-Deeb : A Comparative Study of Game Tree Searching Methods . Thesai [online]. 2018-06-12. [cit. 2018-12-06]. Dostupné z: https://www.researchgate.net/publication/262672371

[2] A GENERAL REINFORCEMENT LEARNING ALGORITHM THAT MASTERS CHESS, SHOGI AND GO THROUGH SELF-PLAY, A general reinforcement learning algorithm that masters chess, shogi and Go through self-play. kstatic.googleusercontent.com [online]. 2021-13-01. [cit. 2021-01-13]. Dostupné z: https://deepmind.com/blog/article/alphazero-shedding-new-light-grand-games-chess-shogi-and-go

[3] CHESS PROGRAMMING WIKI, Razoring. chessprogramming.org [online]. 2020-06-12. [cit. 2020-12-06]. Dostupné z: https://www.chessprogramming.org/Razoring

[4] CHESS PROGRAMMING WIKI, Null Move Pruning. chessprogramming.org [online]. 2020-06-12. [cit. 2020-12-06]. Dostupné z: https://www.chessprogramming.org/Null_Move_Pruning

[5] COMPUTER CHESS, Complete rating list. computerchess.org.uk [online]. 2020-06-12. [cit. 2020-12-06]. Dostupné z: http://www.computerchess.org.uk/ccrl/4040/rating_list_all.html

[6] EMIL FREDRIK ØSTENSEN, A Complete Chess Engine Parallelized Using Lazy SMP. UNIVERSITY OF OSLO [online]. 2016-06-12. [cit. 2020-12-06]. Dostupné z: https://www.duo.uio.no/bitstream/handle/10852/53769/master.pdf?sequence=1

[7] STOCKFISH, Stockfish chess engine. GitHub [online]. 2020-06-12. [cit. 2020-12-06]. Dostupné z: https://github.com/official-stockfish/Stockfish

[8] AI.RUG.NL, Matthia Sabatelli1 , Francesco Bidoia , Valeriu Codreanu and Marco Wiering: Learning to Evaluate Chess Positions with Deep Neural Networks and Limited Lookahead. ai.rug.nl [online]. 2020-06-12. [cit. 2020-12-06]. Dostupné z: https://www.ai.rug.nl/ mwiering/GROUP/ARTICLES/ICPRAM_CHESS_DNN_2018.pdf

[9] CHESS PROGRAMMING WIKI, Frutinity pruning chessprogramming.org [online]. 2020-06-12. [cit. 2020-12-06]. Dostupné z: https://www.chessprogramming.org/Futility_Pruning

[10] CHESS PROGRAMMING WIKI, Parallel search chessprogramming.org [online]. 2020-06-12. [cit. 2020-12-06]. Dostupné z: https://www.chessprogramming.org/Parallel_Search

[11] CHESS PROGRAMMING WIKI, Stockfish chessprogramming.org [online]. 2020-06-12. [cit. 2020-12-06]. Dostupné z: https://www.chessprogramming.org/Stockfish