VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

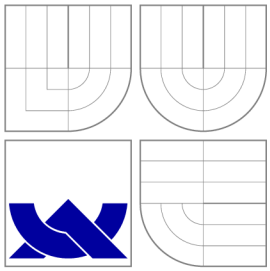
ŠACHOVÝ PROGRAM S RŮZNÝMI VARIANTAMI ŠACHŮ S ROZDÍLNÝM POČÁTEČNÍM ROZESTAVENÍM FIGUR

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

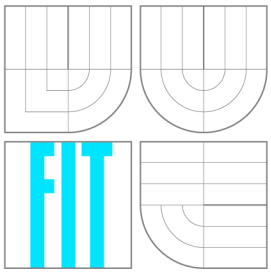
AUTOR PRÁCE
AUTHOR

PAVEL DVOŘÁK

BRNO 2011



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

ŠACHOVÝ PROGRAM S RŮZNÝMI VARIANTAMI ŠACHŮ S ROZDÍLNÝM POČÁTEČNÍM ROZESTAVENÍM FIGUR

CHESS PROGRAM WITH VARIOUS CHESS VARIATIONS WITH VARIOUS INITIAL ARRANGEMENT

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

PAVEL DVOŘÁK

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. JAROSLAV ROZMAN, Ph.D.

BRNO 2011

Abstrakt

Tato práce se zabývá rozborem všech komponent potřebných pro implementaci moderního šachového programu. Cílem je implementace několika šachových variant za použití struktur a algoritmů používaných v profesionálních šachových programech. Práce se zabývá principem reprezentace šachovnice v počítači a faktory hodnocení stavu hry jak v klasickém šachu, tak v implementovaných variantách. Nakonec obsahuje popis a srovnání rozhodovacích algoritmů a jejich rozšíření.

Abstract

This thesis focuses on describing required components in the process of creation of modern chess application. Goal is to create chess program with several chess variations using structures and algorithms based on professional chess programs. Thesis describes principles of chessboard representation and various factors of chessboard state evaluation used in classic chess and implemented variants. Finally thesis describes game-tree search algorithms and enhancements and compares their effect.

Klíčová slova

Šachy, šachové varianty, umělá inteligence, bitboard, Alpha-Beta, transpozice

Keywords

Chess, chess variations, artificial intelligence, bitboard, Alpha-Beta, transposition

Citace

Pavel Dvořák: Šachový program s různými variantami
šachů s rozdílným počátečním
rozestavením figur, bakalářská práce, Brno, FIT VUT v Brně, 2011

Šachový program s různými variantami šachů s rozdílným počátečním rozestavením figur

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana doktora Jaroslava Rozmana.

Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Pavel Dvořák
13. května 2012

Poděkování

Děkuji panu doktorovi Rozmanovi za poskytnuté rady a trpělivost při konzultacích. Dále děkuji rodině a přátelům, kteří mě při tvorbě práce podporovali.

Obsah

Kapitola 1

Úvod

Cílem této práce je popsání běžně používaných postupů, algoritmů a struktur při tvorbě šachového programu a implementace vlastního šachového programu s vybranými variantami za využití získaných znalostí.

Druhá kapitola práce má úvodní charakter. V první polovině se věnuji základním pravidlům šachu, abych mohl následně ve druhé polovině snadněji nastínit rozdíly v pravidlech jednotlivých implementovaných variant.

Třetí kapitola je věnována metodám reprezentace šachu na počítači. Popisuje potřebné datové typy, srovnání algoritmů a nakonec metody generování proveditelných tahů.

Ve čtvrté kapitole popisuji všechny prvky související s hodnocením stavu šachovnice. První polovina je opět věnována obecnému hodnocení klasického šachu, na který poté v druhé polovině navazuji rozborem rozdílů v hodnocení variant.

Pátá kapitola popisuje využití tohoto hodnocení v rozhodovacích algoritmech. Postupně zde popisuji vývoj algoritmu od základního Minimax, přes Alpha-Beta, až po implementaci složitějších rozšíření.

V šesté kapitole jsou nakonec stručně popsány vlastnosti výsledné aplikace, potřebné knihovny a informace o proběhlém testování.

Kapitola 2

Pravidla šachu

2.1 Pravidla klasického šachu

Implementované varianty mají základ v klasickém šachu. Proto nejprve rozeberu pravidla klasického šachu, aby byly odlišnosti variant snadné na pochopení.

2.1.1 Figury a jejich pohyblivost

Implementované varianty šachu se skládají z šesti různých typů figur.

- **Pěšec** je spotřebním materiálem každé strany. Obvykle se používá k rozbití obrany protivníka i za cenu vlastních ztrát. Pěšci se mohou pohybovat o jedno pole vpřed a útočí o jedno pole vpřed na diagonálách. V klasickém šachu a některých variantách se ale na pěšce vztahuje i několik výjimek popsaných dále.
- **Jezdec** je převážně útočná figura. Jako jediný umožňuje přeskakovat jiné figury a tím může protivníka ohrožovat i za jeho obrannou linií. Jezdcům se často daří uzamknout protivníka do "vidličky", tedy ohrožovat alespoň dvě důležité figury najednou.
- **Střelec** se pohybuje neomezeně daleko, ale pouze po diagonálách. Proto se dále dělí na černého a bílého střelce.
- **Věž** se také pohybuje neomezeně daleko, ale pouze horizontálně a vertikálně. Protože tímto může protivníkova krále naprosto odstřihnout od dané části šachovnice, tak je věž cennější než střelec.
- **Dáma** kombinuje pohyblivost věže a střelce, tedy neomezený pohyb po diagonálách, horizontálně i vertikálně. Dáma je nejcennější figurou a její neopodstatněná ztráta často znamená prohru daného hráče.
- **Král** je chráněnou figurou. Nelze jej ztratit, protože jakékoliv ohrožení(*Šach*) vyžaduje bezpodmínečné přesunutí krále do bezpečné pozice nebo odstranění hrozby. Pokud se daný hráč s ohrožením krále nemůže nijak vypořádat, tak prohrává. Král se může pohybovat o jedno pole v libovolném směru, nesmí se ale přesunout na pole ohrožené protivníkem.

2.1.2 Zakončení hry

Hra může skončit jedním ze dvou stavů:

- **Mat** nastane, pokud král hráče na tahu je ohrožený některou figurou protivníka a daný hráč nemůže toto ohrožení nijak zvrátit(přesunutím krále na bezpečné pole, postavením jiné figury útočníkovi do cesty, nebo odstraněním útočníka). Hráč v tomto případě prohrává.
- **Pat** nastane, pokud hráč na tahu nemůže provést žádný platný pohyb některou figurou, ale jeho král není v přímém ohrožení. Patová situace je ekvivalentem remízy.

2.1.3 Speciální tahy

Rošáda

Rošáda je metoda skrytí krále do bezpečí rohových polí šachovnice, a zároveň vystavení jedné z věží z rohu ke středu. Principem je přisunutí krále směrem k vybrané věži, a poté přesunutí věže kolem krále. Rošáda má striktní pravidla pro provedení:

- Králem ani vybranou věží nebylo pohnuto.
- Dráha mezi králem a vybranou věží není blokována žádnou figurou.
- Král není v šachu.
- Žádné z polí na dráze krále při rošádě není ohrožováno protivníkem.
- Král se po dokončení rošády nenachází v šachu.

Podle vybraného směru se rozlišuje malá a velká rošáda. Malá rošáda(královské křídlo) se provádí z pohledu bílého hráče napravo od krále. Pojmenování je podle počtu polí mezi králem a věží(zde pouze dvě). Velká rošáda se provádí na opačné straně(dámské křídlo – tři pole mezi králem a věží).

Tah o dvě pole

Jde o výjimku v pohyblivosti pěšců. Bílí pěšci na druhé řadě a černí pěšci na sedmé řadě mohou provést pohyb o dvě pole dopředu. Takový tah mohou provést pouze v případě, že je dráha mezi pěšcem a cílovým polem volná.

Braní mimochodem

Braní mimochodem(z francouzského *en passant*, doslova "během míjení"[21]) souvisí s předchozí výjimkou a těží z něj naopak druhá strana. Pokud hráč provede tah pěšcem o dvě pole a mine přitom protivníkova pěšce na vedlejším sloupci, tak může protivník tohoto pěšce sebrat jako kdyby provedl normální tah o jedno pole. Tato příležitost platí jen pro protivníkův následující tah, v dalších tazích ji již nelze uplatnit.

Proměna pěšce

Pěšci, kterým se podaří proniknout až na poslední protější řadu(bílí pěšci na osmou řadu, černí pěšci na první řadu), mohou provést proměnu. Proměna umožňuje daného pěšce vyměnit za silnější figuru(dámu, věž, střelce, nebo jezdce). Dáma bývá volena nejčastěji, občas může být ale výhodnější jezdec, pokud k proměně dojde v situaci, která vede k výhodné "vidličce". Nemá příliš smysl volit věž nebo střelce, protože jejich tahy jsou pouze podmnožinou tahů dámy. Hráč je často volí v případě, že figuru hned po proměně stejně ztratí.

2.1.4 Další pravidla

Trojité opakování

Pravidlo trojitého opakování uvádí, že pokud se na začátku tahu některého hráče hra nachází v pozici, která již byla alespoň třikrát zopakována, tak hráč na tahu může vyhlásit pat. K trojitému opakování nejčastěji dochází pří opakovaném šachu. V takové situaci je jediným smysluplným tahem jednoho hráče ohrožení protivníkova krále. Protivník na to musí reagovat přesunutím krále do bezpečí, ovšem má k dispozici pouze pole, které bude ohroženo vzápětí. Bez pravidla by tedy došlo k nekonečnému přesouvání útočící figury a krále po čtveřici polí.

Pravidlo padesáti tahů

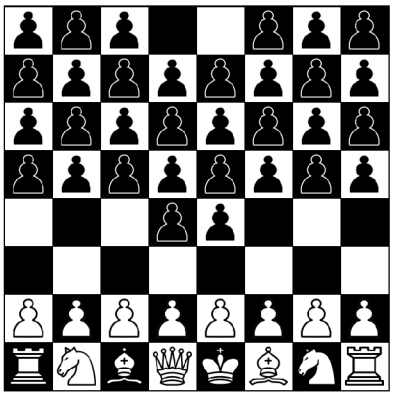
Toto pravidlo také souvisí se zabráněním vzniku nekonečné hry. Podle tohoto pravidla dochází k patu po uplynutí padesáti po sobě jdoucích tahů, během kterých nedošlo k sebrání žádné figury a nebylo pohnuto žádným pěšcem.

2.2 Popis implementovaných variant

2.2.1 Horde

Jde o modifikaci varianty, kterou vytvořil Lord Dunsany roku 1942 [20]. V původní verzi disponoval bílý hráč třemi řadami pěšců, a stál proti standardní sestavě figur černého hráče. V modifikaci Horde bílý hráč disponuje standardní sadou figur se stejným rozmístěním jako v klasickém šachu. Černý hráč hraje pouze s pěšci, zato jich má k dispozici celkem 32. Rozmístění figur na počátku hry je znázorněno na obrázku 2.1. Tato varianta umožňuje proměnu pěšců obou hráčů na první, resp. osmé řadě. Pohyb pěšců o dvě pole je možný pouze z druhé a sedmé řady. Ačkoliv vypadá množství černých pěšců na první pohled děsivě, tak ve skutečnosti je černý hráč i tak v nevýhodě. Jakmile na osmou řadu pronikne dáma nebo věž, tak černý hráč rychle prohrává. Tomu odpovídají i statistiky, podle kterých černý hráč zvítězí pouze ve 25% her [7].

- **Podmínky vítězství bílého hráče:** Bílý hráč zvítězí, pokud se mu podaří sebrat všechny pěšce černého hráče.
- **Podmínky vítězství černého hráče:** Černý hráč zvítězí, pokud dá šachmat bílému králi.
- **Podmínky patové situace:** Některý z hráčů nemůže provést žádný platný tah. To se v této variantě může především stát černému hráči, protože pohyblivost pěšců je nízká a je snadné jejich možný pohyb v nepozornosti zablokovat.

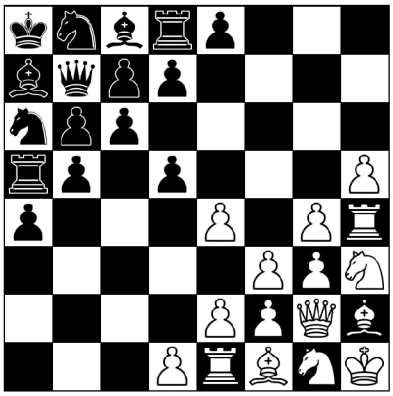


Obrázek 2.1: Varianta Horde

2.2.2 Legan

V této variantě jsou figury hráčů umístěny zrcadlově do protilehlých rohů(obrázek 2.2). Počty figur zůstávají stejné. Varianta neumožňuje rošády, ale proměna pěšců je stále možná na třech polích vertikálně a horizontálně směrem od krále a na poli krále(dva krajní pěšci nemají přímou možnost proměny). Pěšci nemohou provést tah o dvě pole(a tedy ani provést sebrání "mimochodem"). Hlavní změnou v pravidlech je způsob tahu pěšců. Pěšci se zde pohybují na následující políčko po diagonále směrem k protivníkovi a útočí na vedlejší políčko v horizontálním a vertikálním směru. Způsob pohybu pěšců je tedy jakoby otočen o 45°.

- **Podmínky vítězství bílého hráče:** Bílý hráč zvítězí, pokud dá šachmat černému králi.
- **Podmínky vítězství černého hráče:** Černý hráč zvítězí, pokud dá šachmat bílému králi.
- **Podmínky patové situace:** Některý z hráčů nemůže provést žádný platný tah.

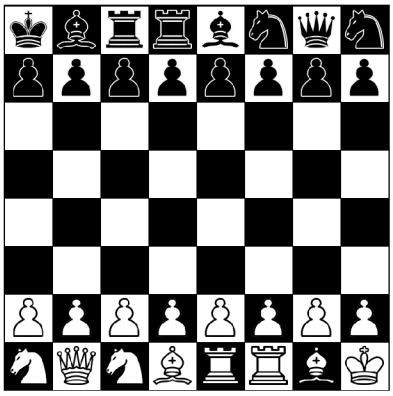


Obrázek 2.2: Varianta Legan

2.2.3 Corner

Pojmenování této varianty vychází z umístění králů do rohu šachovnice(obrázek 2.3). V rozmístění pěšců není změna, ale ostatní figury jsou umisťovány náhodně na svoji řadu. Pouze u střelců platí pravidlo, že jeden střelec musí být umístěn na černé pole, druhý na bílé pole. Protivníkovy figury jsou poté umístěny zrcadlově. Varianta neuvažuje rošádu, ale proměna pěšců, pohyb o 2 pole a metoda braní "mimochodem"možné jsou.

- **Podmínky vítězství bílého hráče:** Bílý hráč zvítězí, pokud dá šachmat černému králi.
- **Podmínky vítězství černého hráče:** Černý hráč zvítězí, pokud dá šachmat bílému králi.
- **Podmínky patové situace:** Některý z hráčů nemůže provést žádný platný tah.



Obrázek 2.3: Varianta Corner

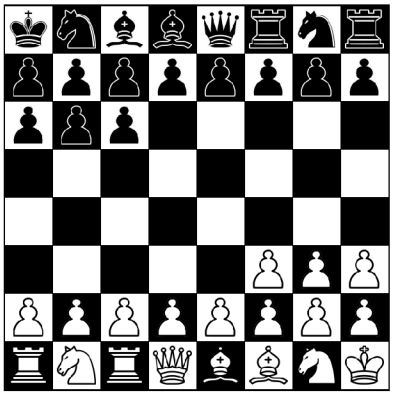
2.2.4 Fortress

Všechny pravidla a rozmístění jsou stejná jako ve variantě Corner. Modifikací je přidání tří pešců na pozice před krále(obrázek 2.4). Tím je král umístěn do své "pevnosti".

- **Podmínky vítězství bílého hráče:** Bílý hráč zvítězí, pokud dá šachmat černému králi.
- **Podmínky vítězství černého hráče:** Černý hráč zvítězí, pokud dá šachmat bílému králi.
- **Podmínky patové situace:** Některý z hráčů nemůže provést žádný platný tah.

2.2.5 Massacre

Jde o největší odchýlení od původních pravidel a variantu oblíbenou především u méně zkušených hráčů. Jde v podstatě o šachovou obdobu karetní hry Válka. V této variantě má každý z hráčů k dispozici 8 jezdců, 8 střelců, 8 věží a 8 dam. Jde o 32 figur pro každého hráče, dohromady 64. Je tedy zaplněno každé pole na šachovnici(obrázek 2.5). Těchto 64



Obrázek 2.4: Varianta Fortress

figur je na šachovnici rozmístěno zcela náhodně, a tak šance na vítězství závisí i na štěstí daného hráče. Je zde jedno důležité pravidlo. Jediné povolené tahy jsou útočné. Nelze se pouze přesunout na pole, je třeba sebrat protivníkovu figuru. Pravidla pohyblivosti všech figur jsou stejné, jako v klasickém šachu. Žádné speciální tahy ovšem možné nejsou.

- **Podmínky vítězství bílého hráče:** Bílý hráč zvítězí, pokud sebere všechny černé figury, nebo černý hráč nemůže provést žádný platný tah.
- **Podmínky vítězství černého hráče:** Černý hráč zvítězí, pokud sebere všechny bílé figury, nebo bílý hráč nemůže provést žádný platný tah.
- **Podmínky patové situace:** Patová situace zde nenastává.

Obrázek 2.5: Varianta Massacre

Kapitola 3

Šachy na počítači

3.1 Reprezentace stavu hry

Z hlediska efektivity a rychlosti programu je způsob reprezentace dat velice důležitý. Snahou je, aby se často prováděné operace, jako je například vyhledání všech možných tahů, prováděly co nejrychleji. Zpočátku byly používány pole 64 bytů, kde každý byte reprezentoval jedno hrací pole a jeho stav(zda se na daném poli nachází nějaká figura, o jakou figuru jde, jaké je barvy...). Další variantou bylo pole pouze 32 bytů, kde každý byte odpovídal jedné figuře a opět informoval o jejím typu, stavu a navíc i souřadnicích. Tyto struktury se zpracovávaly výhradně sekvenčními algoritmy, což bylo pomalé. Na druhou stranu vyžadovaly malé množství paměti, a tak byly aplikovatelné i na slabších zařízeních. Revoluci v reprezentaci stavu šachovnice způsobil takzvaný *bitboard*.

3.1.1 Bitboard

Tato metoda byla pravděpodobně poprvé použita v roce 1950, kdy ji Arthur Samuel použil ve své implementaci hry Dáma [22]. Metoda byla poté v roce 1959 popsána v článku "*Some Studies in Machine Learning Using the Game of Checkers*"svazku "*IBM Journal of Research and Development*". Pro hru Šachy byl bitboard poprvé použit v roce 1960 v Sovětském svazu týmem, který pracoval na šachovém programu KAISSA. Tento program běžel na 64b mainframe, a protože šachovnice obsahuje 64 polí, tak bylo možné reprezentovat stav v jediném paměťovém bloku.

Bitboard využívá pole 64 bitů, respektive jeden 64b integer. Každý bit v tomto čísle odpovídá jednomu poli šachovnice a může nabývat hodnoty 0 nebo 1. Můžeme tedy například zjistit jestli je některé pole obsazeno(1), nebo je prázdné(0). Mapování jednotlivých indexů na pole šachovnice se různí podle implementace. Ve své aplikaci využívám Little-Endian notaci, takže nejméně významný bit(index 0) odpovídá poli A1 a nejvíce významný bit(index 63) odpovídá poli H8(tabulka 3.1). Jediné takové číslo ale pro reprezentaci celého stavu hry nestačí. Pro reprezentaci stavu šachové partie se používá celkem 12 bitboardů, každý obsahuje umístění jednoho typu figur pro každou barvu(př. tabulka 3.2). Jde tedy o 6 pro bílého hráče(pěšci, koně, střelci, věže, dáma a král) a dalších 6 pro černého hráče.

Hlavní výhodou takové reprezentace je, že umožňuje provádění bitových operací, které jsou z hlediska procesoru velmi rychlé. Předpokladem je využití na 64b CPU, kde je možné takovou bitovou operaci provést v jediné instrukci.

Například pro provedení tahu(tabulka 3.3) stačí provést operaci:

```
(současný_stav XOR původní_nová_pozice)
```

8	56	57	58	59	60	61	62	63
7	48	49	50	51	52	53	54	55
6	40	41	42	43	44	45	46	47
5	32	33	34	35	36	37	38	39
4	24	25	26	27	28	29	30	31
3	16	17	18	19	20	21	22	23
2	8	9	10	11	12	13	14	15
1	0	1	2	3	4	5	6	7
	A	B	C	D	E	F	G	H

Tabulka 3.1: Reprezentace jednotlivých polí v bitboardu.

Pro zjištění, zda je dané pole obsazené, slouží operace:

```
(kontrolované_pole AND (bílé_figury OR černé_figury))
```

Posledním příkladem je jednoduchost zjištění, zda figura ohrožuje krále. To je provedeno pomocí operace:

```
(možné_tahy_figury AND bitboard_krále)
```

Bez použití bitboardu by bylo nutné sekvenčně prohledávat jednotlivá pole, resp. figury, a kontrolovat, zda náhodou tah nekončí na stejném poli, na kterém je král.

Díky využití bitových operací je navíc možné využívat při provádění tahu vlastnosti operace XOR, kdy při zopakování operace dojde k návratu na původní hodnotu. To je velice praktické pro rozhodovací algoritmy a sekvenci `proveď_tah`, `prohledej_podstrom`, `vrať_tah`.

0x000000000000ff00

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1
0	0	0	0	0	0	0	0

Tabulka 3.2: Bitboard pro bílé pěšce.

3.1.2 Bitcount algoritmus

První pomocnou funkcí, potřebnou pro práci s bitboardy, je Bitcount algoritmus [10]. Tento algoritmus vrátí počet bitů, které jsou v předaném bitboardu nastaveny na 1. Tento algoritmus se hodí zejména u ohodnocovací funkce, protože můžeme z 64b čísla snadno a rychle získat počet figur daného hráče, a ten poté ohodnotit.

0x0000000000020200

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0
0	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Tabulka 3.3: Bitboard, znázorňující tah bílým pěšcem z B2 na B3.

Naivní přístup

Nejtriviálnější přístup je sekvenční průchod po bitech a zaznamenávání stavu nejméně významného bitu(LSB). Tento postup vyžaduje konstantní čas, ale je oproti ostatním metodám pomalý, protože vyžaduje vždy 64 operací bez ohledu na stav bitboardu. Je možné jej optimalizovat přidáním kontroly, zda bitboard není nulový, ale i tak bude většinou pomalejší než následující metody(viz tabulka 3.4).

```
int popCount_brute (bitboard x)
{
   int count = 0;
   for (int i = 0; i < 64; i++, x >>= 1)
      count += (int)x & 1;
   return count;
}
```

Algoritmus Briana Kernighana

Tento algoritmus v cyklu resetuje nejméně významný bit daného bitboardu a počítá počty průchodů cyklem. Cyklus je ukončen, pokud je bitboard nulový. Rychlost tohoto algoritmu závisí na počtu nastavených bitů. Jeho použití se vyplatí, pokud neočekáváme více nastavených bitů než 10. Pro více jak 10 nastavených bitů je lepší využití algoritmu HAKMEM. U více jak 52 nastavených bitů je tento algoritmus pomalejší než sekvenční průchod po jednotlivých bitech(viz tabulka 3.4).

```
int popCount_BK (bitboard x)
{
   int count = 0;
   while (x)
   {
       count++;
       x &= x - 1; // reset LS1B
   }
   return count;
}
```

Algoritmus MIT HAKMEM

Algoritmus [6] postupně rozděluje bitboard pomocí masky na posloupnosti 2, 4, 8, 16 a 32 bitů. V každé posloupnosti spočítá počet bitů nastavených na hodnotu log. 1 a tímto počtem danou posloupnost přepíše. Výsledku je dosaženo sečtením jednotlivých produktů bitového maskování. Výhodou je konstantní čas algoritmu a absence větvení, takže procesor nemusí řešit předpovídání větvení [17]. Pokud očekáváme malý počet nastavených bitů, tak je rychlejší využití algoritmu BK(viz tabulka 3.4).

```
int bitCount_HAKMEM(bitboard board)
{
    bitboard  M1 = 0x5555555555555555ull;  // 1x0,1x1 ...
    bitboard  M2 = 0x3333333333333333ull;  // 2x0, 2x1 ...
    bitboard  M4 = 0x0f0f0f0f0f0f0f0full;  // 4x0, 4x1 ...
    bitboard  M8 = 0x00ff00ff00ff00ffull;  // 8x0, 8x1 ...
    bitboard M16 = 0x0000ffff0000ffffull;  // 16x0, 16x1 ...
    bitboard M32 = 0x00000000ffffffffull;  // 32x0, 32x1 ...

    board = (board & M1 ) + ((board >>  1) & M1 );
    board = (board & M2 ) + ((board >>  2) & M2 );
    board = (board & M4 ) + ((board >>  4) & M4 );
    board = (board & M8 ) + ((board >>  8) & M8 );
    board = (board & M16) + ((board >> 16) & M16);
    board = (board & M32) + ((board >> 32) & M32);
    return (int)board;
}
```

Tento algoritmus je možné využít pro počítání počtu nastavených bitů i v jiných než 64b číslech. Zde je ilustrace průběhu výpočtu u 4b čísla na základě předchozího pseudokódu [23]:

```
M1 = 0101 // 1x0, 1x1
M2 = 0011 // 2x0, 2x1
B = 1111 // vstup - 4x1

// první operace - počet bitů nastavených na log. 1
// v každé dvojici bitů je uložen do této dvojice
B = (B & M1) + ((B >> 1) & M1)
```

$$\begin{aligned}(B \wedge M1) + ((B \gg 1) \wedge M1) &= B \\ (1111 \wedge 0101) + ((0111) \wedge 0101) &= 1010\end{aligned}$$

```
B = 1010
// druhá operace - počet bitů nastavených na log. 1
// v každé čtveřici bitů je uložen do této čtveřice
B = (B & M2) + ((B >> 2) & M2)
```

$$\begin{aligned}(B \wedge M2) + ((B \gg 2) \wedge M2) &= B \\ (1010 \wedge 0011) + ((0010) \wedge 0011) &= 0100\end{aligned}$$

```
B = 0100 // výsledek - 4 bity nastavené na log. 1
```

Vzorek(počet log. 1)	Brute	BK	HAKMEM
0x0000000000000001 (1)	3,311s	0,148s	0,679s
0x0000000000000cff (10)	3,268s	0,655s	0,678s
0x5555555555555555 (32)	3,208s	2,107s	0,693s
0xfffffffffffff000 (52)	3,268s	3,293s	0,677s
0xffffffffffffffff (64)	3,316s	4,166s	0,676s

Tabulka 3.4: Srovnání Bitcount algoritmů pro 10 000 000 průchodů.

3.1.3 Bitscan algoritmus

Bitscan algoritmy slouží k získání indexů bitů v log. 1 ze zdrojového bitboardu. Díky tomu můžeme například skenováním bitboardu bílých pěšců získat všechny jejich pozice, podle kterých poté generujeme jednotlivé možné tahy. Stejně tak možné tahy každé figury jsou nejprve v podobě bitboardu, ze kterého přesná políčka musíme extrahovat.

Naivní přístup

Podobně jako u algoritmu Bitcount je možné postupně testovat všech 64 bitů a ukládat indexy bitů v log. 1. Protože šachové bitboardy bývají řídce osazené, tak bývá efektivnější indexování pomocí vyhledávání nejméně významného bitu.

Algoritmus De Bruijnovy posloupnosti

Tento algoritmus vyvynul v roce 1997 Martin Läuter [8]. Algoritmus využívá De Bruijnovu posloupnost k vytvoření hash klíče, pomocí kterého se přistupuje do tabulky indexů.

Definice 3.1 *Nechť A je abeceda symbolů délky k.* De Bruijnova posloupnost $B(k, n)$ *je taková cyklická posloupnost, kde každá n-tice symbolů z abecedy A v posloupnosti nastane právě jednou.*

Algoritmus využívá posloupnost $B(2, 6)$, tedy abecedu dvou symbolů (0,1) a cyklická posloupnost je tvořena všemi unikátními 6-ticemi symbolů z této abecedy. Nejvyšších 6 bitů posloupnosti je poté použito jako index do tabulky s pozicemi hledaných nejméně významných bitů.

Princip je nastíněn v následujícím pseudokódu [23]:

```
int firstBit(bitboard board)
{
    int INDEX64[64] = {     // pozice nejméně významných bitů
        63,  0, 58,  1, 59, 47, 53,  2,
        60, 39, 48, 27, 54, 33, 42,  3,
        61, 51, 37, 40, 49, 18, 28, 20,
        55, 30, 34, 11, 43, 14, 22,  4,
        62, 57, 46, 52, 38, 26, 32, 41,
        50, 36, 17, 19, 29, 10, 13, 21,
        56, 45, 25, 31, 35, 16,  9, 12,
        44, 24, 15,  8, 23,  7,  6,  5
    };
```

```
... pokračování
    // De Bruijnova posloupnost B(2,6)
    bitboard DEBRUIJN64 = 0x07EDD5E59A4E28C2;

    // izolace LSB
    int lsb = (board & -board);

    // výsledný index LSB je získán z tabulky levým posuvem posloupnosti o
    // násobek určený izolovaným LSB, a poté izolováním nejvyšších 6 bitů
    // pomocí pravého posuvu o 58b
    return INDEX64[(lsb*DEBRUIJN64)>>58];
}
```

Algoritmus je efektivní pro řídce osazené bitboardy. Pokud je osazeno více bitů za sebou, tak se efektivita hledání LSB snižuje. Proto je možné implementovat různé kombinace sekvenčního průchodu a vyhledávání LSB. Srovnání sekvenčního průchodu a vyhledávání LSB je znázorněno v tabulce 3.5. Zvyšování času u sekvenčního průchodu(který by jinak měl být konstantní) je způsobeno přidáváním indexu do pole pro návrat.

Vzorek(počet log. 1)	Brute	De Bruijn
0x0000000000000001 (1)	0,948s	0,645s
0x0000000000000cff (10)	2,817s	2,674s
0x5555555555555555 (32)	4,956s	5,286s
0xffffffffffffffff (64)	7,19s	8,313s

Tabulka 3.5: Srovnání algoritmů Bitscan pro 1 000 000 průchodů.

3.2 Generování možných tahů

Pro generování tahů se používají jako základ předpočítané bitboardy. Je jich poměrně velké množství(pro každou variantu tahu je jich 64 – možnosti pohybu pro každé pole), z toho vychází větší paměťová náročnost než u jiných řešení, ale při implementaci pro počítače nepředstavuje paměť velký problém. Velkou výhodou je totiž rychlé nalezení všech možných tahů, snadná kontrola stavu šachovnice a jednoduché prohledávání stavového prostoru pro potřeby umělé inteligence. Předpočítané tahy neberou v úvahu žádná obsazená pole, tento test se provádí až v samotném algoritmu generování tahů. Jedinou restrikcí jsou tedy hranice šachovnice. Každý z bodů následujícího seznamu představuje pole o 64 prvcích, ve kterém je každým prvkem 64b integer(bitboard). Tento bitboard označuje možnost pohybu vybrané figury. Na pole v log. 1 lze táhnout, na pole v log. 0 nikoliv. Indexování je shodné s číslováním šachovnice, věž na poli A1(index 0) tedy ke svým tahům přistupuje jako `smer_tahu[0]`. Tahy jsou v mé implementaci předpočítány následovně:

- **Pěšci**
  - Bitboard pro tahy o jedno pole.
  - Bitboard pro tahy o dvě pole(postup ze druhé a sedmé řady).
  - Bitboard pro útok(pole na diagonále vlevo a vpravo).

  - Bitboard pro kontrolu braní mimochodem(pole vlevo a vpravo).
  - Bitboard pro pozici proměny pěšců(kvůli implementaci dalších variant, ve kterých tato pozice nemusí být první a poslední řada).

- **Jezdec**
  - Stačí jedna sada bitboardů pro pohyb z každého pole.

- **Střelec**
  - Bitboard pro pohyb po diagonále vlevo nahoru.
  - Bitboard pro pohyb po diagonále vlevo dolů.
  - Bitboard pro pohyb po diagonále vpravo nahoru.
  - Bitboard pro pohyb po diagonále vpravo dolů.

- **Věž**
  - Bitboard pro pohyb vlevo.
  - Bitboard pro pohyb vpravo.
  - Bitboard pro pohyb nahoru.
  - Bitboard pro pohyb dolů.

- **Dáma** jako jediná nemá žádné předpočítané tahy, protože její tahy lze získat logickým součtem tahů střelce a věže.

- **Král**
  - Stačí jedna sada bitboardů pro pohyb z každého pole.

3.2.1 Jednoduché tahy

Pěšci, král a jezdec se řadí mezi jednoduché tahy. Mohou táhnout pouze o jedno pole(s výjimkou tahu o dvě pole u pěšců, ale to lze jednoduše kontrolovat), a toto pole je, nebo není volné(jednoduchá kontrola logickým součinem s prázdnými poli šachovnice).

3.2.2 "Klouzající"figury

Klouzající figury(střelec, věž, dáma), tedy figury, které se v daných směrech mohou pohybovat dokud nenarazí na nějakou překážku, vyžadují složitější přístup. Nejčastěji se používá jedna z následujících metod:

Klasický přístup

Klasický přístup [9] využívá bitscan algoritmy pro nalezení LSB nebo MSB v závislosti za směru prohledávání. Nalezený bit označuje první překážku v cestě.

Shifted bitboards

Jde o metodu, kterou využívá šachový program NagaSkaki(viz. [16]) a kterou jsem implementoval i ve vlastním šachovém programu. Místo vyhledávání LSB a MSB využívá série bitových posuvů a sjednocení k získání platné sekvence polí. Kromě předgenerovaných bitboardů pro jednotlivé pohyblivosti nevyžaduje žádné další tabulky, avšak vyžadují poměrně velké množství operací pro získání výsledku. Tahy věže nebo střelce jsou získány pomocí 56 bitových operací [15].

Příklad postupu získání tahů věže vpravo:

- Provedeme logický součin tahů věže vpravo a obsazených polí. Získáme jednotlivá obsazená pole v cestě věže.

```
right_moves = right_board[sq] AND occupiedboard
```

- Postupně provedeme bitový posuv o 1 až 6 bitů a všechny tyto posuvy sjednotíme. Získáme tak všechna pole, na která věž nemůže táhnout. První obsazené pole v cestě totiž způsobí nastavení všech zbývajících na hodnotu log. 1.

```
right_moves = (right_moves<<1) OR (right_moves<<2) OR (right_moves<<3)
      OR (right_moves<<4) OR (right_moves<<5) OR (right_moves<<6)
```

- Protože bitový posuv může přetéct i do jiných řad, tak musíme provést logický součin současného stavu a možných tahů věže vpravo.

```
right_moves = right_moves AND right_board
```

- Získali jsme tedy pole, na která v pravém směru na dané řadě věž táhnout nemůže. Pro získání polí, na která věž táhnout může, stačí provést operaci XOR současného stavu a možných tahů věže vpravo. Nakonec musíme provést sjednocení s prázdnými poli a poli, ovládanými protivníkem. Vyloučíme tak pokus o sebrání vlastní figury.

```
right_moves = (right_moves XOR right_board) AND enemy_and_empty_squares
```

Princip pro zbývající směry věže a pro střelce je naprosto stejný, mění se pouze směr a velikost bitového posuvu. U pohybu nahoru a dolů jde o násobky osmi, u diagonál o násobky sedmi a devíti.

Rotated bitboards

Tato metoda využívá stav řady jako 6b index do tabulky platných tahů [19]. Řady samy o sobě mají 8 bitů, ale protože dva krajní bity nemají na pohyb žádný vliv(pokud se na krajním poli nachází vlastní figura, tak na pole táhnout nelze a pokud je na poli figura protivníka, tak ji lze vyhodit), tak je možné je vynechat. Tím se velikost tabulky sníží z 256 na 64. Jde tedy o několik 64x64 maticí obsahujících bitboardy s platnými tahy pro každou kombinaci stavu na dané řadě. Samotný 6b index je ze stavu řady získán maskováním a poté bitovým posuvem tak, aby byl mezi nejméně významnými bity.

V předchozím textu jsem psal pouze o řadách, pro tahy věží jsou ale potřeba i sloupce a pro střelce i pohyb vertikálně. Pro ty se používají otočené bitboardy. Pokud stav ve sloupci otočíme o 90°, tak získáme stav v řadě, ze kterého opět po bitovém posuvu získáme index do tabulky. Stejný princip je použit i pro tahy střelců, kde je otočení o 45°(diagonální vektory 8x8 matice budou po otočení představovat jednotlivé řady). Tohoto otočení dosáhneme změnou indexace matice.

Magic bitboards

Magické bitboardy [18] jsou v moderních šachových programech pravděpodobně nejrozšířenější. Ve své podstatě vycházejí z otočených bitboardů tím, že také používají stav šachovnice jako index. Pro převod ze sloupce a diagonály na řadu ale používají metodu vynásobení bitboardu "magickým"číslem, které způsobí namapování stavu hry na hash, který je opět použit jako index do tabulky platných tahů(tabulka 3.6).

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	E	0
0	0	0	0	0	D	0	0
0	0	0	0	C	0	0	0
0	0	0	B	0	0	0	0
0	0	A	0	0	0	0	0
0		0	0	0	0	0	0

$*$(magické číslo) =

*	*	*	A	B	C	D	E
*	*	*	*	*	*	*	*
*	*	*	*	*	*	*	*
*	*	*	*	*	*	*	*
*	*	*	*	*	*	*	*
*	*	*	*	*	*	*	*
*	*	*	*	*	*	*	*

*	*	*	A	B	C	D	E
*	*	*	*	*	*	*	*
*	*	*	*	*	*	*	*
*	*	*	*	*	*	*	*
*	*	*	*	*	*	*	*
*	*	*	*	*	*	*	*
*	*	*	*	*	*	*	*

$>> (64 - 5) =$

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
A	B	C	D	E	0	0	0

Tabulka 3.6: Princip Magických bitboardů.

Magická čísla nejsou pevně stanovená. Platným magickým číslem je libovolné číslo, které způsobí namapování stavu relevantních sloupců, řad a/nebo diagonál na posloupnost po sobě jdoucích bitů. Je tedy možné je odhadnout sérií pokusů a omylů, nebo použít brute-force algoritmy testující náhodná čísla. Vygenerovaná magická čísla je také možné nalézt na internetu [11].

Zatímco u generování založeného na bitových posuvech stačí udržovat tabulky možných tahů z daného pole až na konec šachovnice(na platné tahy je transformována až dodatečně), tak u magických(i otáčených) bitboardů musíme v tabulce udržovat všechny možné bitové kombinace pro každé pole. Místo 64x64b(512B) na směr tedy tabulky obsahují 64x64x64b(32kB) na směr. Vzhledem k tomu, že u stolních počítačů nebývá paměť problémem a že toto řešení umožňuje získání všech platných tahů v konstantním čase jen na základě přístupu do tabulky, tak je toto řešení preferováno. Ve své aplikaci jsem ale ponechal metodu posuvů, protože rozdíl v rychlosti nebyl dle mého měření(porovnání rychlosti nalezení tahu v šesti scénářích ukazuje tabulka 3.7) příliš velký.

3.2.3 Generování legálních tahů

Při generování legálních tahů jsou postupně pro každou figuru vygenerovány možné tahy, a tyto tahy jsou následně testovány, zda neponechávají krále daného hráče v šachu. Tahy, které v testu neuspějí, jsou z množiny tahů odstraněny.

Scénář	Shifted	Magic	Střelci	Věže	Dámy
1	6,76s	5,833s	3	4	2
2	2,41s	2,286s	2	0	2
3	0,392s	0,369s	1	2	0
4	0,211s	0,181s	2	0	0
5	5,398s	4,842s	3	1	2
6	3,153s	2,835s	4	4	2

Tabulka 3.7: Srovnání metod Shifted a Magic bitboard.

3.2.4 Generování pseudo-legálních tahů

Protože testování legálnosti každého tahu je časově náročné, tak se toto testování vynechává a algoritmus generování tahů vrací pouze pseudo-legální tahy. V rozhodovací funkci je poté třeba tah na legálnost testovat. Tato metoda počítá s tím, že dojde k redukci stromu a velké množství tahů bude díky tomu vynecháno. Je proto zbytečné testovat na legálnost úplně všechny tahy.

3.2.5 Řazení tahů

Řazení tahů zvyšuje šanci, že nejlepší tah bude prohledáván jako jeden z prvních. V implementované aplikaci je řazení z důvodu rychlosti provedeno na okně relevantních tahů pomocí klasického insert sort algoritmu. Zbytek tahů je ponechán v pořadí, v jakém byly tahy vygenerovány. Relevantní tahy jsou řazeny následovně:

1. **Předchozí nejlepší tah**
   Používá se hlavně ve spojení s Iterative deepening algoritmem. Nejlepší tahy získané z předchozího průchodu jsou zařazeny na první místo v následujícím průchodu.

2. **History moves**
   Za předchozí nejlepší tahy jsou řazeny takové tahy, které byly někdy v minulosti vyhodnoceny jako kvalitní, nebo způsobující redukci stromu. Pokud je implementovaný některý hash algoritmus, tak máme tyto tahy uloženy v transpoziční tabulce. Bez transpoziční tabulky bývají tyto tahy ukládány do menších polí(například jeden až dva nejlepší tahy pro každou hloubku, nebo několik tahů způsobujících redukci).

3. **Útoky – MVV-LVA**
   Jde o zkratku fráze *Most Valuable Victim – Least Valuable Aggressor*. Principem je seřazení útočných tahů na základě hodnoty útočníka a oběti. Čím hodnotnější je oběť a méně hodnotný útočník, tím dříve v poli je takový tah zařazen. Útoky, ve kterých pěšec sebere dámu, jsou tedy řazeny jako první, útoky dámy na pěšce naopak jako poslední.

Kapitola 4

Ohodnocení stavu hry

4.1 Obecný pohled

Člověk, hrající šachy, si stále pokládá otázku "Je pro mě tento tah dobrý?". K tomu, aby mohl tuto otázku zodpovědět, musí nejdříve provést zhodnocení stavu hry, který by po provedení tahu nastal. Pro potřeby umělé inteligence je třeba zavést heuristickou ohodnocovací funkci, která na základě přijatých dat o stavu šachovnice vrátí číselnou hodnotu. Tato hodnota nese informaci o souhrnném stavu hry a zejména o poměru sil. Obecně může toto ohodnocení dosahovat tří důležitých stavů:

- **Nulová hodnota** říká, že tah vede k vyváženému stavu hry.
- **Kladná hodnota** vyjadřuje převahu hráče na tahu. Tato převaha může být dána větším počtem figur, nebo lepší pozicí.
- **Záporná hodnota** naopak vyjadřuje materiálovou a/nebo poziční nevýhodu oproti soupeři.

Tato hodnota je získána součtem dílčích analýz, kterým se budu věnovat dále.

4.1.1 Cena figur

Nejzákladnější způsob ohodnocování je na základě dostupného materiálu. Jde tedy o součet počtu jednotlivých typů figur a vynásobení vhodnou konstantou, reprezentující cenu tohoto typu figury(tabulka 4.1). Toto ohodnocení hraje velkou roli v celkovém ohodnocení stavu hry a umělá inteligence s vhodně zvoleným oceněním figur je schopna odehrát poměrně uspokojivé partie. Hodnocení krále se zpravidla nepoužívá při hodnocení stavu hry, protože krále nelze ztratit. Jeho hodnocení spíše slouží jako vyjádření ohodnocení vyhrané nebo prohrané partie. Základem pro jakékoliv ocenění jak figur, tak pozic, bývá ohodnocení jednoho pěšce. Vše ostatní je poté od hodnoty jednoho pěšce odvozeno(tzv. *centipawn*). Pokud jeden pěšec má hodnotu 100, tak věž s hodnotou 500 vyjadřuje, že je 5x hodnotnější než pěšec. Dosažená pozice ohodnocená na 20 bodů má hodnotu $\frac{1}{5}$ pěšce. Není tedy důležité, jakou absolutní hodnotu jsme přiřadili, ta se bude u jednotlivých implementací lišit. Vše závisí pouze na poměru hodnocení figur a pozic.

Umělá inteligence, používající pouze faktor ceny figur, se vyznačuje následovně:

- V začátku hry provádí první možný tah, protože většina tahů vyústí v nulové skóre.
- Využívá možností sebrání cenné figury, zároveň se brání sebrání vlastních figur.

Pěšec	100
Jezdec	320
Střelec	325
Věž	500
Dáma	900
Král	40000

Tabulka 4.1: Použité ocenění figur v normální hře.

- Není schopná odhalit budoucí nebezpečí plynoucí z výhodné pozice na šachovnici. Dříve nebo později podlehne vidličce způsobené jezdcem či střelcem uprostřed šachovnice.

Z předchozích bodů je patrné, že vyhodnocení podle ceny figur je dostatečné proti začátečníkovi, ale i občasný hráč nebude mít s poražením takové umělé inteligence příliš velké potíže.

Je vhodné cenu figur upravovat v průběhu partie. Jezdec je důležitý hlavně zpočátku, protože pro protivníka není snadné se vyhnout jeho útoku a je velká pravděpodobnost, že jezdec ohrozí více figur jedním tahem. S ubývajícími figurami ale tuto výhodu postupně ztrácí. Proto může být vhodné jej v závěru ohodnocovat menší cenou než na začátku partie. Střelci mají opačnou vlastnost. Zpočátku mají velmi omezenou pohyblivost a nejsou příliš užiteční. S ubývajícími figurami ale jejich kontrola šachovnice vzrůstá. Proto se jejich cena v závěru může o něco zvýšit. Navíc dva střelci se navzájem doplňují, tudíž může být brán v úvahu i bonus za zachování obou střelců. Je důležité podotknout, že má význam měnit skóre až v závěru, kdy figura získá nebo ztratí svoji užitečnost. Pokud by se inverzní operace provedla na začátku, tak by to mohlo umělou inteligenci vést k obětování figury, která může být později velmi důležitá.

4.1.2 Pohyblivost figur

Vyjadřuje počet polí, na která v daném tahu může hráč táhnout. Každé takové pole ve své podstatě omezuje možnosti tahu protivníka, protože ten by se tak vystavil riziku sebrání. Nicméně problémem hodnocení kontroly hracího pole je fakt, že většina takových tahů je bezvýznamná a jen minimum opravdu blokuje protivníka. Program by mohl mít tendence rozbít své obranné pozice jen kvůli přesunu na pole, které poskytuje velkou kontrolu šachovnice. Proto je lepší hodnotit pozice jednotlivých figur z pohledu předpokládané pohyblivosti na těchto pozicích. Například jezdci nejsou blokováni žádnými figurami a jsou tak výborným nástrojem ke kontrole soupeře a pronikání za jeho linie. Proto jejich preferovaným umístěním je střed šachovnice, kde se předpokládá největší pohyblivost a tím také užitečnost. Střelci a věže mohou upřednostňovat pozice, na kterých není jejich pohyb v některém směru blokován vlastními figurami nebo ohraničením šachovnice. Rohy, kraje a případně i startovní pole bývají z toho důvodu hodnoceny nízkým nebo přímo záporným počtem bodů.

4.1.3 Kontrola šachovnice

Na rozdíl od prosté pohyblivosti(všechna pole, na která daná figura může vykonat validní tah) kontrola šachovnice vyjadřuje, která pole jsou bezpečně chráněna před napadením.

Pole je kontrolováno daným hráčem v případě, že je bráněno více figurami, než s kolika může oponent pole napadnout.

4.1.4 Pozice krále

Pro případ nutnosti přesunu krále je vhodné vyjádřit, která pole jsou vhodná pro přesun více, a která méně. Obecně nejméně vhodná pole jsou v rozích šachovnice, protože odtud má král pouze 3 možnosti tahu, všechny snadno blokovatelné. Ideální pole jsou taková, která poskytují maximální počet možných tahů. Toto tvrzení ale platí zejména pro závěr hry. Zpočátku by umělá inteligence měla klást důraz naopak na ochranu krále. Pokud šachová varianta umožňuje provedení rošády, tak může být její provedení odměněno ohodnocením daných polí. Také je možné sledovat, jak velký tlak je v daném tahu na krále vyvíjen. Nejjednodušší metodou je prostá vzdálenost figury od krále, čím kratší tato vzdálenost je, tím více je král "ohrožen".

4.1.5 Pozice pěšců

Při ohodnocování pěšců bereme v úvahu několik typů pěšců: volní pěšci, krytí volní pěšci, zdvojení pěšci, izolovaní pěšci a blokovaní pěšci.

Volní pěšci nejsou ve svém postupu na protější řadu blokováni žádným pěšcem protivníka(žádný protivníkův pěšec není ve směru pohybu na stejném sloupci jako pěšec hráče). Tito pěšci jsou ve variantách, umožňujících změnu figury při dosažení určitého místa na šachovnici, velmi důležití. Jako takoví jsou ohodnocováni větším počtem bodů a jejich ochrana by měla být mezi prioritami. Změna pěšce například v dámu poskytuje obrovskou výhodu.

Krytí volní pěšci jsou volní pěšci, kteří jsou chráněni některou figurou. Toto má význam zejména v závěru hry, kdy se hlavní útočnou figurou stane král. Ten jako jediný nemůže sebrat chráněnou figuru, proto zajištění krytí volných pěšců je velice důležité.

Zdvojení pěšci jsou dva(v horším případě i více) pěšci stejné barvy v jednom sloupci. Tento stav zpravidla nastane na začátku hry, kdy se pěšci navzájem vyhazují. Jde o poměrně nežádoucí jev, protože se pěšci navzájem blokují v postupu a je dobré se buďto takovým tahům vyhýbat, nebo zdvojené pěšce přesunout na některý volný sloupec.

Izolovaní pěšci mohou být i volní pěšci. Jsou jimi takoví pěšci, kteří na sousedních sloupcích nemají žádné jiné pěšce stejné barvy. Jejich ochrana tedy může být zajištěna pouze hlavními figurami.

Blokovaní pěšci jsou pěšci, zastavení v postupu po sloupci protivníkovým pěšcem. Jde o nepříjemný stav, protože takový sloupec je až do uvolnění blokován a tím znesnadňuje pohyb po šachovnici ostatních figur.

Nakonec u pěšců počítáme ještě s jedním faktorem. Je jím plný stav pěšců. Zpravidla se sice snažíme zachovat co nejvíce figur, ale plný stav pěšců blokuje pohyb hlavních figur, zejména věží. Proto je vhodné plný stav penalizovat a tím se snažit alespoň jeden sloupec uvolnit.

4.2 Rozdíly v implementovaných variantách

4.2.1 Horde

Ve variantě Horde disponuje Bílý hráč standardní sestavou figur, kdežto Černý hráč má k dispozici 32 pěšců.

Bílý hráč se pokouší získat kontrolu nad co největším prostorem na ploše a vyčkává, než Černý hráč posune některého pěšce do nechráněné pozice. K prolomení počáteční obrany Černého hráče dobře poslouží pěšci. Důležitý je postup na osmou řadu s věžemi a dámou, který umožní rychlou výhru, pokud zůstane první řada chráněná. U hodnocení bílých pěšců jsem snížil hodnocení postupu. Je nepravděpodobné, že by se bílí pěšci dostali až na osmou řadu a proto je lepší, když zůstanou v obraně a vytvoří trhliny v řadách pěšců protivníka.

Černý hráč postupuje obezřetně vpřed s pěšci. Toho je možné dosáhnout postupně se zvyšující hodnotou řad směrem k bílému hráči. Použil jsem hodnocení $\frac{1}{100}$ až $\frac{1}{10}$ ceny pěšce. Tím jsem dosáhl toho, že AI postupuje na první řadu opravdu spíše pomalu. Důležité v ohodnocení je zajištění vzájemné ochrany pěšců. Hodnotu penalizace nechráněných pěšců jsem zvolil větší než hodnotu postupu na libovolnou další řadu. Umělá inteligence má tedy za cíl postupovat na první řadu aby dosáhla proměny pěšců, ale ne za cenu vystavení vlastního pěšce do nechráněné pozice.

4.2.2 Legan

Protože je zde rozmístění otočeno o 45°, tak je třeba upravit většinu pozičních hodnocení. Pro pěšce jsem použil podobné postupné hodnocení jako v klasickém šachu, jen místo řad jsou inkrementálně hodnoceny diagonály. Penalizoval jsem postup do rohů a na kraje šachovnice, kde nedochází k proměně, protože na těchto polích jsou pěšci ve slepé uličce a jen omezeně přispívají kontrole šachovnice. Podobně jako se v klasickém šachu ohodnocuje postup věže na sedmou řadu pro útoky na pěšce, tak jsem ve variantě Legan tento bonus přidal střelcům pro útok na diagonály pěšců.

4.2.3 Corner

Ve své podstatě není třeba hodnocení příliš měnit. Kvůli náhodném rozmístění figur jsem z hodnocení odebral postihy za setrvání v počáteční pozici a nahradil je obecným hodnocením nevýhodnosti dané pozice. Podobně jsem u krále odstranil bonusy související s rošádou.

4.2.4 Fortress

Oba hráči mají v rohu svého krále k dispozici "pevnost"ze tří pěšců navíc. Tito pěšci mohou být využiti buďto pro obranu, nebo pro rychlý útok na slabší křídlo protivníka a vybojování proměny některého pěšce. Proto jsem zvýšil hodnocení postupu pěšců po křídle s pevností. Ostatní rozdíly jsou stejné jako ve variantě Corner.

4.2.5 Massacre

V Massacre šachu získává vysokou prioritu pohyblivost figur daného hráče. Jedním z pravidel totiž je, že jediné povolené tahy jsou takové, kterými hráč sebere figuru protivníka. Umělá inteligence tedy musí vykonávat prioritně takové tahy, které umožní velký počet

následně možných tahů. Naopak se musí vyvarovat tahů do slepých uliček, protože takové tahy mohou vyústit v nepoužitelnost dané figury po zbytek hry a potenciálně i prohru. Proto jsem zvolil pro každý platný tah poměrně vysoké skóre, a to $\frac{1}{2}$ ceny pěšce.

Oceňování pozice jsem neřešil. Vzhledem k náhodnosti celé hry může být kterákoliv pozice dobrá, pokud umožňuje velký počet následných tahů.

Z hlediska ceny figur není třeba velkých změn. Největší cenu má stále dáma, protože má největší pohyblivost a tedy největší pravděpodobnost, že bude umožňovat další tahy. Jezdec je problematický, protože u jeho tahů záleží na štěstí nebo smůle hráče. Může se velice snadno stát, že některý jezdec nebude mít žádnou možnost tahu. S ubývajícími figurami možnosti Jezdce klesají mnohem rychleji, než u ostatních figur. Proto u jezdce bude záležet především na ocenění počtu následných tahů, oproti jeho vlastní ceně.

Kapitola 5

Rozhodovací algoritmy

5.1 Úvod

5.2 Minimax

Základním algoritmem pro hraní strategických her je algoritmus Minimax. Je založen na principu výběru nejlepšího možného tahu s předpokladem, že protivník na tento tah zareaguje opět pro sebe tím nejlepším možným tahem. Protože člověk zpravidla nehraje ideálně, tak tento postup poskytuje umělé inteligenci určitou výhodu. Hlavními faktory jsou poté správné ohodnocení stavu hry a dostatečná hloubka průchodu stromem. S hloubkou průchodu stromem souvisí velká nevýhoda algoritmu Minimax. Tento algoritmus prochází naprosto všechny možnosti tahu umělé inteligence i protivníka a nebere ohled na fakt, že některé podstromy protivník nebude brát vůbec v úvahu, protože již nejlepší tah nalezl. Proto se Minimax běžne rozšiřuje o algoritmus redukce Alpha-Beta.

5.2.1 Pseudokód

```
minimax(max, hloubka)
{
    if(hloubka == 0)
        return HodnoceniStavu()
    else
    {
        foreach(seznam_tahů)
        {
            provedTah();
            hodnoceni = minimax(max,hloubka-1)
            vratTah();
            if(hodnoceni > max)
                max = hodnocení
        }
        return max
    }
}
```

5.3 Alpha-Beta

Alpha-Beta [5][1] rozšiřuje Minimax o možnost ořezání irelevantních podstromů. Tím je v průměru docíleno zrychlení algoritmu. Algoritmus udržuje interval, ve kterém je hodnocení daného tahu relevantní. Dolní hranice tohoto intervalu označuje hodnotu nejlepšího tahu, na který hráč na tahu narazil. Cokoliv s hodnocením menším než dolní hranice je horší tah. Horní hranice označuje maximální ohodnocení, které bude protivník brát v úvahu. Tahy s vyšším hodnocením, než je horní hranice, jsou příliš dobré a protivník si je kvůli existující alternativě nevybere. Změna hráče na tahu je ve vyhledávací funkci simulována prohazováním hodnot alpha a beta. Díky tomu je stále udržován aktuální interval `<nejlepší_tah,maximální_tah>`.

Příklad: *Uvažujme tah, ve kterém Bílý hráč táhne dámou na pole, na kterém sebere koně Černého hráče. V tomto momentu se to zdá jako výborný tah. Nyní algoritmus začne prohledávat možné reakce na tento tah. První tah je tah pěšcem na volné pole, který se vyhodnotí až do dané hloubky. Druhý tah je tah střelcem na volné pole a opět se vyhodnotí do dané hloubky. Třetí tah je tah střelcem, který zapříčiní sebrání dámy Bílého hráče. Pokud v tomto momentu tah způsobí pro Bílého hráče značnou nevýhodu, tak nemá smysl prohledávat další tahy, protože víme, že Černý hráč v nejlepším případě bere Bílou dámu a tomu se chce Bílý hráč vyhnout.*

hloubka	**maximum**	**minimum**
n	b^n	$b^{(n/2)} + b^{(n/2)} - 1$
1	40	40
2	1,600	79
3	64,000	1,639
4	2,560,000	3,199
5	102,400,000	65,569
6	4,096,000,000	127,999
7	163,840,000,000	2,623,999
8	6,553,600,000,000	5,119,999

Tabulka 5.1: Počty zkoumaných uzlů v různých hloubkách prohledávání při použití Alpha-Beta. [14]

Jak je vidět v tabulce 5.1, tak může dojít ke značnému snížení počtu uzlů v rozhodovacím stromě bez ztráty přesnosti. Pouze se odstraní ty podstromy, které na výsledek nebudou mít žádný vliv(jednoduše řečeno algoritmus oznámí, že tento tah je špatný a už může být jen horší). Hodnoty v tabulce používají jako základ průměrný počet platných tahů. Použitý vzorec $b^{(n/2)} + b^{(n/2)} - 1$ vychází z předpokladu, že ty nejlepší tahy jsou v seznamu umístěny na první pozici, a k oříznutí tedy dojde hned na začátku. V praxi ovšem předem nevíme, který z tahů je ten nejlepší, protože právě pro zjištění nejlepšího možného tahu je tento algoritmus aplikován. Můžeme ale dosáhnout určitého odhadu nejlepšího tahu za pomocí různých metod řazení tahů, případně využitím metody Iterative Deepening a nebo tahů uložených v transpoziční tabulce.

5.3.1 Pseudokód

```
alphaBeta(alpha,beta,hloubka)
{
   if( hloubka == 0 )
      return HodnoceniStavu()

   foreach( pseudolegalni_tahy )
   {
      provedTah()
      // kontrola validity pseudolegálních tahů
      if( kralOhrozen(strana) )
      {
         vratTah()
         continue
      }
      score = -alphaBeta(-beta, -alpha, hloubka-1)
      vratTah()

      if( score >= beta )
         return beta
      if( score > alpha )
         alpha = score
  }
  return alpha
}
```

5.3.2 Typy uzlů

V algoritmu Alpha-Beta rozlišujeme následující typy uzlů [12]:

PV-Node

Hodnocení těchto uzlů se nachází v intervalu (α, β). Jde tedy o vylepšení předchozího nejlepšího tahu. Tyto uzly jsou prozkoumávány celé a jsou zpravidla v nějaké podobě ukládány pro pozdější potřebu řazení.

Cut-Node

Hodnocení těchto uzlů je větší než β. Tyto uzly reprezentují tah, který je příliš dobrý. Protivník tedy již nalezl tah, který jej staví do lepší situace a tento podstrom by tak nezvolil. Tyto uzly způsobují redukci stromu.

All-Node

Hodnocení těchto uzlů je menší než α. Při prohledávání byl tedy již nalezen lepší tah. V těchto uzlech je prohledán celý podstrom.

5.4 Další rozšíření

5.4.1 Horizon effect

Pojem *Horizon effect*, tedy *"Efekt horizontu"*, vyjadřuje neschopnost umělé inteligence vidět ve svých tazích za určitou hranici, tedy horizont [2]. Tato hranice je dána hloubkou prohledávání. Ve vyhodnocování nejlepšího možného tahu hraje tento jev velkou roli. Šachové programy mohou vykonávat tahy, které s prohledávanou hloubkou vypadají pozitivně, ale "za horizontem"je nebezpečí ztráty materiálu nebo rovnou prohry. Toto nebezpečí je ale pro umělou inteligenci skryto.

Příklad: *Algoritmus Alpha-Beta pro Bílého hráče vyhodnotí sebrání Černého koně Bílou dámou jako nejlepší tah v rámci dané hloubky* `n`*. Umělá inteligence tedy tento tah provede. Ovšem možnou reakcí v hloubce* `n+1` *bylo sebrání Bílé dámy Černým pěšcem, což už algoritmus nemohl vyhodnotit.*

Tento problém není možné úplně odstranit, protože jsme limitování výpočetním výkonem počítače a z něj vyplývajícím limitem hloubky. Je ale možné jej některými metodami zmírnit. Mezi takové metody patří Check extension a Quiescence search.

5.4.2 Check extension

Tato metoda se aplikuje na dvě rozdílné situace:

- Hráčův král dostal šach.
- Hráč dal šach králi protivníka.

V obou situacích je prohledávaná hloubka zvýšena o 1, aby byla pokryta i možná reakce na tuto událost. Protože počet možných odpovědí na šach je limitován, tak nehrozí velké zvýšení časové náročnosti.

5.4.3 Quiescence search

Nějakou podobu metody *Quiescence search* používají v podstatě všechny moderní šachové programy [3]. Název metody je odvozen z principu vyhledávání a ohodnocování pouze "tichých"(*Quiet*) pozic. Jde o pozice, ve kterých nehrozí velká taktická změna ve hře(ztráta materálu, šach...). Základním principem je, že pokud je pozice "tichá", tak je možné ji ohodnotit. Pokud není, tak se pokračuje v prohledávání podstromu této pozice až do nalezení "tiché"pozice.

Narozdíl od *Check extension* se kontrolují navíc i pozice, ve kterých dojde ke ztrátě materiálu. Toto v závislosti na typu hry může způsobit explozi stavů. V klasickém šachu to nebývá problém, ale například v implementované variantě Massacre by aplikace Quiescence search vždy způsobila prohledání celého stavového prostoru. Proto ji v této variantě nevyužívám a spoléhám se na klasický Alpha-Beta algoritmus a ohodnocení počtu platných tahů.

Pseudokód

```
int Quiesce( int alpha, int beta )
{
    // spojení Check extension s Quiescence search
    if( kralOhrozen(strana) )
        return alphaBeta( alpha, beta, hloubka=1 )
    int score = HodnoceniStavu();
    if( score >= beta )
        return beta;
    if( alpha < score )
        alpha = score;
    foreach( utocne_tahy )
    {
        provedTah();
        score = -Quiesce( -beta, -alpha );
        vratTah();
        if( score >= beta )
            return beta;
        if( score > alpha )
           alpha = score;
    }
    return alpha;
}
// úprava Alpha-Beta algoritmu
alphaBeta( alpha, beta, hloubka )
{
   if( hloubka == 0 )
   {
      //return HodnoceniStavu()
      return Quiesce( alpha, beta )
   }


   (...)
```

5.4.4 Transpoziční tabulka

Transpoziční tabulka [4] je hashovací tabulka, do které jsou ukládány vyhodnocené tahy pro pozdější využití. Záznamy v této tabulce se hodí pro řazení tahů, nebo získání předchozího hodnocení tahu.

Transpozice

Šachová transpozice je jeden stav materiálu na šachovnici, ke kterému lze dojít různou sekvencí tahů. Protože je ohodnocován právě finální stav a ne postup, kterým jsme se do tohoto stavu dopracovali, tak je výhodné ukládat ohodnocení tohoto stavu. Pro reprezentaci a ukládání transpozice do transpoziční tabulky potřebujeme nějakým způsobem jednoznačně a jednoduše vyjádřit stav hry. K tomu potřebujeme některou hashovací funkci.

Zobrist hashing

Jde o techniku hashování, sloužící k téměř jednoznačné reprezentaci stavu šachovnice ve formě čísla. Technika je pojmenována podle svého autora, kterým je Albert Zobrist [13].

Stav šachovnice je reprezentován 64b klíčem. Zobrist hashing je založený na podobném principu, jako provádění tahů na bitboardech. Při každé změně stavu šachovnice je provedena na klíči operace XOR s jiným 64b klíčem, který reprezentuje danou změnu. Je tedy možné jednoduše provádět a vracet změny.

Klíče, reprezentující změny, jsou pseudonáhodně vygenerovány při startu aplikace. Jejich počet se může lišit podle implementace, v zásadě jde ale o následující:

- 64x12 klíčů pro index pole na šachovnici a typ figury pro obě strany(64 polí, 6 bílých typů, 6 černých typů)
- Klíč pro změnu strany na tahu.
- Minimálně 16 klíčů pro tahy *en passant*.
- Čtyři klíče pro jednotlivé rošády.

Příklad provedení tahu bílé věže z A1 na A3:

```
soucasny_hash = soucasny_hash XOR zobrist_pohyb[A1][BILA_VEZ]
soucasny_hash = soucasny_hash XOR zobrist_pohyb[A3][BILA_VEZ]
soucasny_hash = soucasny_hash XOR zobrist_zmena_strany
```

Postupným prováděním operace XOR nad hashem při každé změně stavu na šachovnici můžeme udržovat unikátní číselnou hodnotu tohoto stavu. Máme tedy k dispozici klíč pro transpoziční tabulku a zároveň unikátní hodnotu pro každý stav šachovnice. Ve spojení s historií tahů můžeme tedy navíc kontrolovat, zda nedochází k opakování některých tahů.

5.4.5 Iterative deepening

Jde o metodu procházení stavového prostoru, při které se provádí série prohledání do omezené hloubky, která se postupně zvyšuje. Nejlepší tah z předchozího průchodu je při následujícím průchodu zařazen na začátek. Metoda má formu cyklu `for`, ve kterém opakovaně voláme algoritmus Alpha-Beta se zvyšující se hloubkou prohledávání a ukládáme získané tahy. Může se zdát, že bude prostor mnohokrát prohledáván zbytečně, ale ve spojení s transpoziční tabulkou nejde o velkou časovou ztrátu, neboť můžeme od určité hloubky použít hodnocení uzlů z předchozího průchodu. Navíc ziskem je velice slušný odhad nejlepšího možného tahu.

5.4.6 Null move pruning

Jde o heuristiku sloužící k časné detekci uzlů, které způsobí redukci stromu. Heuristika je založena na teorii, že libovolný tah by měl způsobit zlepšení pozice daného hráče. Pokud tedy zkusíme vzdát se svého tahu a i tak vrácené skóre způsobí redukci stromu(je větší než beta), tak to znamená že se nacházíme z pohledu protivníka v příliš silné pozici a provedení tahu by pozici jen posílilo. Je tedy zbytečné prohledávat celý podstrom a můžeme vrátit hodnotu rovnou. Tato heuristika je provedena před normálním Alpha-Beta průchodem a provádí volání algoritmu Alpha-Beta do velice omezené hloubky(přibližně o 4 menší než normálně použitá hloubka). Alpha-beta interval je zúžen okolo hodnoty beta, takže dojde k velkému počtu redukcí.

```
if(umoznit_null && !kralOhrozen(strana) && dostatekMaterialu(strana))
{
   score = -alphaBeta(-beta,-beta+1,hloubka-NULL_REDUKCE);
   if(score > beta)
      return score;
}
```

Samozřejmě se může stát, že se hráč naopak nachází v takové pozici, že neprovedení žádného tahu je pro něj nejlepší varianta, nebo se nachází v situaci, kdy nějaký tah provést musí. Takové stavy je třeba v heuristice ošetřit, protože už samotné uvažování přeskočení tahu odporuje pravidlům a heuristika by poté mohla vést ke špatným výsledkům. Zpravidla je tedy provedení heuristiky umožněno pouze pokud jsou splněny následující podmínky:

- Hráčův král není v ohrožení. Při ohrožení krále by vynechání vlastního tahu vedlo k neplatnému stavu šachovnice.
- Hráč má dostatečné množství materiálu. S ubývajícím materiálem se zvyšuje šance výskytu situace, kdy žádný tah je ten nejlepší.

5.4.7 Principal variation search

Jde o podobný algoritmus jako Null move pruning, ve své podstatě zakládá na opačném předpokladu. Zatímco Null move pruning prováděl prohledání za předpokladu příliš silné pozice(vrácená hodnota by měla být větší než beta), tak PV-search provádí prohledání za předpokladu, že nejlepší možná pozice již byla nalezena a vrácená hodnota tedy nebude v aktuálním alpha-beta intervalu. Alpha-beta interval je při volání v tomto případě naopak zúžen okolo hodnoty alpha.

```
if(pv_nalezen)
{
   score = -alphaBeta(-alpha-1,-alpha,hloubka-1);
   // předpoklad nebyl správný, musíme provést celkové prohledání
   if(score > alpha && score < beta)
      score = -alphaBeta(-beta,-alpha,hloubka-1);
}
```

Efektivní použití této metody závisí na kvalitním řazení tahů. Pokud se podaří zařadit nejlepší tah na jednu z prvních pozic, tak díky úzkému intervalu alpha-beta získáme stejný výsledek při větší redukci stromu. Při špatném řazení se ale naopak stane, že budeme prohledávání často opakovat.

Kapitola 6

Implementace

6.1 Popis aplikace

Samotná aplikace je vytvořena pomocí jazyka C++ a knihovny Qt 4.7.0. Disponuje vlastním grafickým uživatelským rozhraním s možností volby vzhledu figur a šachovnice. Umožňuje hru dvou lidských hráčů, lidského hráče a umělé inteligence, nebo dvou umělých inteligencí proti sobě. Pro zjednodušení ovládání rozhraní zvýrazňuje proveditelné tahy. V nastavení lze zapnout i zvýrazňování polí, která jsou ohrožována protivníkem. Každý provedený tah je uložen do historie ve formátu přehledném i pro začátečníky. Tato historie tahů umožňuje i návrat k libovolnému tahu, který byl proveden v minulosti. Hru lze v libovolném okamžiku uložit a z uložené pozice hru opět obnovit. Při obnovení pozice je možné znovu vybrat, zda bude hrát umělá inteligence, nebo člověk. Hra disponuje i nastavitelným časovačem, který omezuje maximální délku tahu každého z hráčů a možností pozastavení hry.

Aplikace byla testována na operačních systémech Windows XP x86, Windows 7 x86/x64 a Ubuntu Linux 11.10 x86.

6.2 Vyžadované knihovny

Aplikace vyžaduje následující součásti pro svoji funkčnost:

- libgcc
- mingw – knihovny potřebné pro překlad
- knihovny Phonon – přehrávání zvuku
- QtCore – základní struktury Qt
- QtGui – grafické součásti Qt aplikace

U verze pro Windows jsou všechny potřebné knihovny přiloženy a aplikace je ihned spustitelná.

Pro generování Makefile, připojení knihoven a následný překlad na systémech Linux je dále potřeba program *qmake*.

Kapitola 7

Závěr

V práci jsem popsal zvolené varianty šachů a rozdíly v pravidlech, které bylo nutno v aplikaci řešit. Na základě zkušeností s ohodnocovacími funkcemi klasického šachu jsem navrhl a popsal potřebné úpravy hodnocení u jednotlivých variant. Pro reprezentaci stavu šachovnice bylo několik voleb, tudíž jsem popsal principy jednotlivých reprezentací, jejich klady a zápory a důvod volby bitových posuvů. Nakonec jsem se věnoval rozhodovacím algoritmům a jejich rozšíření. Zaměřil jsem se především na běžně používaná rozšíření, která jsem následně implementoval v aplikaci, nicméně existuje mnoho jiných, které jsem vynechal(například z důvodu, že byly pouze podmnožinou některého popsaného rozšíření). V příloze A je možné nalézt srovnání přínosu jednotlivých algoritmů na efektivitu umělé inteligence.

Z hlediska dalšího vývoje aplikace přináší tři zajímavé možnosti. V první řadě je vždy možné dál optimalizovat algoritmy umělé inteligence(přechod na Magické bitboardy, další vylepšení řadících funkcí, jiné metody prohledávání stromu). Z hlediska grafiky je dále možné aplikaci rozšířit o moderní 3D rozhraní. Poslední možností je vyřešení absence síťové hry přidáním podpory P2P nebo server-klient komunikace.

Literatura

[1] Baudet, G. M. An analysis of the full alpha-beta pruning algorithm. In ACM New York. *Proceedings of the tenth annual ACM symposium on Theory of computing*. 1. vyd. 1978. S. 296–313.

[2] Frayn, C. a Justiniano, C. The ChessBrain Project. *Studies in Computational Intelligence*. 1. vyd. 2007, roč. 71. S. 91–115. ISSN 978-3-540-72704-0.

[3] Lorenz, U. a Tscheuschner, T. Plyer Modeling, Search Algorithms and Strategies in Multi-player Games. In Springer-Verlag Berlin. *Proceedings of the 11th international conference on Advances in Computer Games*. 1. vyd. 2006. S. 221–223. ISBN 978-3-540-48887-3.

[4] Marsland, T. A. a Campbell, M. A survey of enhancements to the alpha-beta algorithm. In ACM New York. *Proceedings of the ACM '81 conference*. 1. vyd. 1981. S. 109–114. ISBN 0-89791-049-4.

[5] Schaetter, J. New Advances in Alpha-Beta Searching. In ACM New York. *Proceedings of the 1996 ACM 24th annual conference on Computer science*. 1. vyd. 1996. S. 124–130. ISBN 0-89791-828-2.

Webové reference

[6] Anderson, S. E. *Bit Twiddling Hacks* [online]. 2005 [cit. 2012-04-12]. Dostupné na: <http://graphics.stanford.edu/~seander/bithacks.html>.

[7] BrainKing. *Horde chess* [online]. 2009 [cit. 2012-05-07]. Dostupné na: <http://brainking.com/en/GameRules?tp=4>.

[8] Chess Programming. *Bitscan - De Bruijn Multiplication* [online]. 2007, Updated Apr 19 13:36:00 2012 [cit. 2012-04-20]. Dostupné na: <http://chessprogramming.wikispaces.com/BitScan>.

[9] Chess Programming. *Classical Approach* [online]. 2007, Updated Oct 9 01:19:00 2011 [cit. 2012-04-16]. Dostupné na: <http://chessprogramming.wikispaces.com/Classical+Approach>.

[10] Chess Programming. *Population Count - Loop Approaches* [online]. 2007, Updated Apr 3 13:16:00 2012 [cit. 2012-04-20]. Dostupné na: <http://chessprogramming.wikispaces.com/Population+Count#Loop-Approaches>.

[11] Chess Programming. *Best Magics so far* [online]. 2008, Updated Jan 4 09:51:00 2009 [cit. 2012-04-12]. Dostupné na: <http://chessprogramming.wikispaces.com/Best+Magics+so+far>.

[12] Chess Programming. *Node Types* [online]. 2008, Updated Mar 28 11:33:00 2012 [cit. 2012-04-10]. Dostupné na: <http://chessprogramming.wikispaces.com/Node+Types>.

[13] Chess Programming. *Zobrist Hashing* [online]. 2008, Updated Oct 29 01:18:00 2011 [cit. 2012-05-08]. Dostupné na: <http://chessprogramming.wikispaces.com/Zobrist+Hashing>.

[14] Chess Programming. *Alpha-Beta* [online]. 2009, Updated Feb 3 06:17:00 2012 [cit. 2012-04-10]. Dostupné na: <http://chessprogramming.wikispaces.com/Alpha-Beta#Savings>.

[15] Chess Programming. *Shifted Bitboards* [online]. 2011, Updated Jan 9 16:33:00 2011 [cit. 2012-04-16]. Dostupné na: <http://chessprogramming.wikispaces.com/Shifted+Bitboards>.

[16] Mayothi. *How NagaSkaki plays chess* [online]. 2007, Updated Apr 3 13:16:00 2012 [cit. 2012-04-20]. Dostupné na: <http://www.mayothi.com/nagaskakichess6.html>.

[17] Ostrovsky, I. *Fast and slow if-statements: branch prediction in modern processors* [online]. 2010 [cit. 2012-04-10]. Dostupné na: <http://igoro.com/archive/fast-and-slow-if-statements-branch-prediction-in-modern-processors/>.

[18] Rival Chess. *Magic Bitboards* [online]. 2011 [cit. 2012-04-16]. Dostupné na: <http://www.rivalchess.com/magic-bitboards/>.

[19] The Department of Computer and Information Sciences. *Rotated bitmaps, a new twist on an old idea* [online]. [cit. 2012-04-16]. Dostupné na: <http://www.cis.uab.edu/info/faculty/hyatt/bitmaps.html>.

[20] Wikipedia. *Dunsany's chess* [online]. 2007, Updated Aug 9 03:53:00 2011 [cit. 2012-05-08]. Dostupné na: <http://en.wikipedia.org/wiki/Dunsany's_chess>.

[21] Wikipedia. *Braní mimochodem* [online]. 2008, Updated Apr 10 10:46:00 2012 [cit. 2012-04-10]. Dostupné na: <http://cs.wikipedia.org/wiki/En_passant>.

[22] Wikipedia. *Bitboard* [online]. 2012, Updated Mar 4 17:08:00 2011 [cit. 2012-05-08]. Dostupné na: <http://en.wikipedia.org/wiki/Bitboard>.

[23] Wingchess – Winglet. *Winglet chess bitops.cpp* [online]. 2011, Updated Sep 9 2011 [cit. 2012-04-20]. Dostupné na: <http://www.sluijten.com/winglet/08display01.htm>.

Příloha A

Srovnání jednotlivých rozšíření

Následující tabulka obsahuje srovnání efektu jednotlivých rozšíření na rychlost a přesnost algoritmu. Pro odhad přesnosti(hodnota ELO) jsem využil úlohu 6 z automatického hodnotícího nástroje na adrese

http://www.chessmaniac.com/ELORating/ELO_Chess_Rating.shtml.

Umělá inteligence při testování začíná pouze s algoritmem Alpha-Beta v nastavení střední obtížnosti. Poté jsou postupně uvedená rozšíření zapínána. Každý nový řádek tedy představuje uvedené rozšíření + všechny na předchozích řádcích.

Algoritmus	**čas(s)**	**ELO**	**max. hloubka**	**počet uzlů**	**prozkoumáno**
Alpha-Beta	1,37	1000	5	140422	15348
Quiescence	15,421	2500	35	906288	192370
Null-move	8,241	2500	34	438743	101023
PV-search	4,051	2500	24	197418	48909
Transpozice	3,191	2500	24	171618	35585

Tabulka A.1: Srovnání rychlosti jednotlivých rozšíření.

Příloha B

Uživatelská příručka

B.1 Instalace

B.1.1 Windows

Verze pro systémy Windows je přiložena již zkompilovaná se všemi potřebnými knihovnami. Žádná další instalace není třeba.

B.1.2 Linux

Na CD je přiložený .deb balíček, ze kterého je možné provést instalaci.

Pokud je vyžadován překlad ze zdrojových kódů, tak je možné použít jednoduchý instalační skript `install.sh`. Skript funguje za předpokladu, že jsou všechny potřebné knihovny(libqt4-dev, qt4-qmake, libphonon4 a libphonon-dev) nainstalovány.

B.2 Popis rozhraní

B.2.1 Hlavní menu

Hlavní menu sestává ze tří záložek:

- **Nová hra**
  Na této záložce je možné zvolit variantu nové hry a přiřadit hráče k jednotlivým stranám. Na výběr jsou 3 obtížnosti umělé inteligence.

- **Uložit/načíst hru**
  Zde je možné v horní části hru uložit pod libovolným jménem. Je možné uložit pouze zahájenou hru. Všechny hry jsou ukládány do složky Saves. Zbylá část záložky slouží k výběru hry pro obnovení. Vlevo jsou vypsány všechny nalezené soubory ve složce Saves, vpravo jsou poté k dispozici detaily zvolené uložené hry a tlačítka pro obnovení a smazání.

- **Nastavení**
  V nastavení je možné změnit grafiku figur nebo plochy, měnit hlasitost zvuků a zapnout zvýraznění ohrožených polí. Dále je možné nastavit některé vlastnosti předdefinované umělé inteligence. Příliš velké zvýšení hloubky prohledávání ale povede k velké časové náročnosti.

B.2.2 Herní okno

Na levé straně okna se nachází šachovnice. Na horním a dolním okraji se zobrazují sebrané figury. Po najetí na některou figuru na šachovnici se zeleně zobrazí všechny platné tahy. Pokud je v nastavení zapnuto zvýrazňování hrozby, tak se oranžově zvýrazní pole, na která lze táhnout, ale protivník je ohrožuje.

Pro pohyb s figurami je třeba klepnout na příslušnou figuru a táhnout na požadované pole. Aplikace umožní přetáhnutí figury pouze na platné pole. Pokud je figura přetažena na neplatné pole nebo mimo hrací plochu, tak je navrácena na původní pozici.

Na pravé straně hracího okna je zobrazena strana na tahu v podobě obrázku bílého nebo černého pěšce. Pod tímto obrázkem je aktuální čas. Nakonec pod časem je umístěna historie tahů. Je možné se vrátit k libovolnému tahu v této historii, ovšem nelze tahy opět zopakovat. Všechny vrácené tahy jsou ztraceny. Po klepnutí na tlačítko se hra vrátí do stavu po zvoleném tahu. Pokud tedy byl například proveden návrat k tahu bílé figury, tak bude na tahu černý hráč.