VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

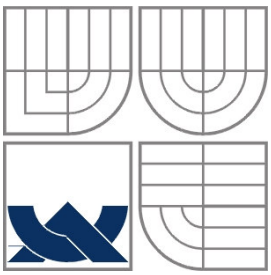
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

APLIKACE SOFT COMPUTING PŘI HRANÍ HER

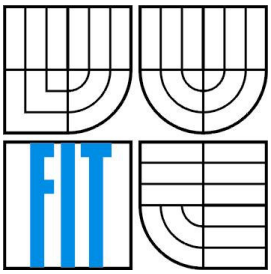
BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE MILAN POSPÍŠIL
AUTHOR

BRNO 2007



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

APLIKACE SOFT COMPUTING PŘI HRANÍ HER

APLICATION OF SOFT COMPUTING FOR GAME PLAYING

BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE
AUTHOR

MILAN POSPÍŠIL

VEDOUCÍ PRÁCE
SUPERVISOR

Doc. Ing. František V. Zbořil, CSc.

BRNO 200

Abstrakt

Tato práce se zabývá aplikací klasických metod umělé inteligence a metod soft computing při hraní her. Pokouší se tyto metody aplikovat na programech pro hraní dámy a šachů a srovnává dosažené výsledky.

Klíčová slova

Umělá inteligence, soft computing, minimax, alfa beta, šachy, dáma.

Abstract

This work deals with aplication of classical methods of artifical intelligence and methods of soft computing for game playing. This methods are applicated in programs for playing chess and draughts. Results are confronted.

Keywords

Artifical intelligence, soft computing, minimax, alfa beta, chess, draughts.

Citace

Milan Pospíšil: Aplikace soft computing při hraní her, bakalářská práce, Brno, FIT VUT v Brně, 2007

Aplikace soft computing při hraní her

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Doc. Ing. Františka V. Zbořila, CSc.

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

……

Jméno Příjmení

Datum

Poděkování

Děkuji za pomoc zejména mému vedoucímu, Doc. Ing. Františku V. Zbořilovi, CSc. za jeho odborné rady.

Obsah

1 Úvod

Umělá inteligence ve hrách je čím dál častěji diskutované téma. Počítačová grafika dosáhla takřka dokonalosti a v současné době není takový tlak na její zlepšení. Daleko lepšího komerčního výsledku dosahují hry s dobrou hratelností, nápady a kvalitní umělou inteligencí. Umělá inteligence je totiž poslední věc, ve které počítačové hry zaostávají. Pochopitelně ve hrách, jako jsou šachy, nebo dáma, je počítač neporazitelný. Ale stačí trochu složitější hra s neurčitostí a počítač nemá šanci.

Až do devadesátých let byla umělá inteligence ve hrách považována za okrajovou záležitost. Avšak s rostoucím zájmem o tyto hry a s rostoucí složitostí her, se hry ukázaly dobrým testem i pro akademický výzkum. Zpočátku se zde aplikovaly hry založené na principech klasické UI - šachy, dáma, reversi apod. Ale u složitějších her nastal problém.
Začala se zde objevovat neurčitost a složitost, že nebylo možné použít prohledávání stavů. Bylo potřeba použít složitějších technik.

U **strategických her** nebyla zpočátku UI příliš dokonalá. Počítač často prostě vytvořil několik jednotek a poslal je do nepřátelské základny. Občas byly některé věci naskriptované předem, ale stále se nedalo mluvit o UI. Jedna věc se zde ovšem vyřešit musela a to **prohledávání optimální cesty terénem.**

Zde se nedal použití klasický jednoduchý algoritmus na prohledávání, jelikož zde bylo potřeba uvažovat se změnou průchodnosti terénu (jednotky ve hře se pohybovaly a mohly zablokovat cestu). Dalším a velkým problémem bylo rozmístění uzlů. Bylo potřeba
po mapě rozmístit uzly tak, aby jich bylo co nejméně a aby se jednotka mezi dvěma uzly bez problémů dostala (tudíž se tam nesmí objevit žádná složitá překážka). Navíc ne vždy měl hráč prozkoumanou celou mapu, takže neměl přístup ke všem informacím. A poslední věc bylo znovupoužití nalezené cesty. Ve hře se vyskytoval velký počet jednotek a bylo potřeba vyřešit dočasné ukládání již nalezené cesty a kontrolu její platnosti.

Další problém byly reakce na události. UI musela vzít v úvahu stav hry (suroviny, budovy, armáda, pozice) a podle toho zareagovat (obrana, útok, přemístění, hledání nových zdrojů). Tyto algoritmy již nemohly běžet na bázi prohledávání stavového prostoru. Zde se uplatnila teorie **expertních systémů.**

Je potřeba dodat, že herní UI se obvykle nesnaží vytvořit nejinteligentnějšího protivníka, který člověka hravě porazí (jako je to třeba u šachů), ale o to, aby byla hra co nejzábavnější. Není například na školu, když UI občas udělá nějakou chybu a naopak překvapí něčím tvořivým. Zkrátka je snaha, aby se herní UI chovala podobně jako člověk se všemi výhodami i nevýhodami. Přispívá to tak k lepší atmosféře ze hry. Díky tomu vstupuje do hry teorie **agentových systémů.**

Agent se snaží hrát tak, jako by to hrál člověk. Tedy má nějaké své cíle, oblíbenou strategii

apod. Jedná tak bez ohledu na to, že by pro něj bylo v dané situaci udělat něco lepšího. O co lépe vypadá, když zbabělý vojevůdce ustoupí a vzdá se, než když si propočítá, že má šanci na vítězství a bude bojovat.

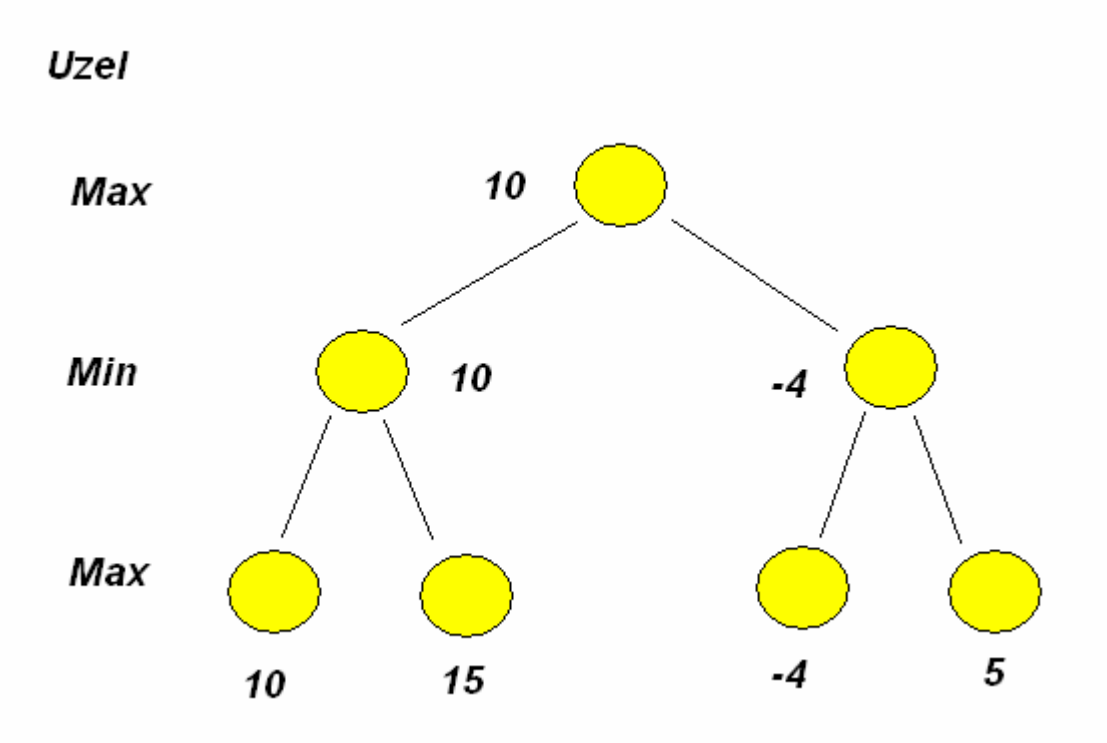
Oblast **genetických algoritmů a neuronových sítí** se zde zatím uchycuje velmi pomalu. Problém je, že zatím osobní počítače nedosahují takové výpočetní síly, aby tyto postupy překonaly postupy klasické. Já osobně jsem se pokoušel tyto metody použít v programu pro hraní šachů a dámy. Neuronové sítě se ukázaly naprosto nepraktické. Genetické algoritmy by šly použít v kombinaci s klasickými metodami. V popisu šachového programu mám toto použití nastíněno.

2 Minimax

Minimax patří mezi klasické metody hraní her. Je velice jednoduchý. Uvedu algoritmus:

Minimax:

1. **Na vstupu máme stav hry X**
2. **Jsme-li v koncovém uzlu (konec hry, či v maximální hloubce), ohodnotíme hru a vrátíme ohodnocení tahu.**
3. **Nejsme-li v koncovém uzlu, generujeme postupně další tahy a voláme tuto funkci rekurzivně s opačným hráčem, než tím na tahu.**
4. **Je-li na tahu hráč, který začal strom tahů generovat, vrať tah s nejlepším výsledkem. Je-li na tahu protihráč, vrať tah s nejhorším výsledkem.**



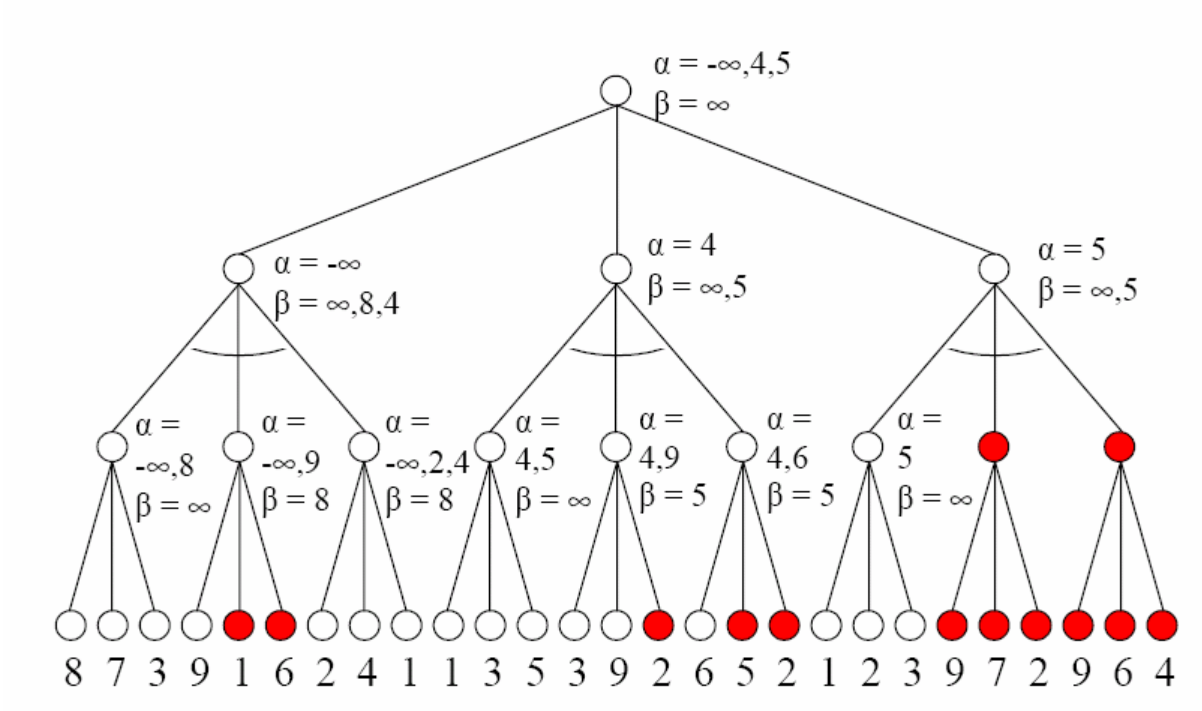
3 Alfa beta

AlfaBeta patří také mezi klasické metody. Je v podstatě pouze vylepšením minimaxu. Vrací totiž stejné výsledky. Vychází z předpokladu, že není třeba prohledávat všechny tahy. V určitých situacích je totiž jasné, že minimax daný tah nemůže vybrat.

Procedura AlfaBeta

1. **Na vstupu máme uzel X**
2. **Je-li X kořenovým uzlem, nastav alfa na min a beta na max**
   (tj na minimální a maximální možnou hodnotu danou rozsahem proměnné)
3. **Je-li X uzlem (tedy koncovým stavem hry, nebo tahem v maximální hloubce), vrať ohodnocení tohoto uzlu.**
4. **Je-li na tahu hráč, který strom generuje, potom:**
   Dokud je alfa < beta potom generuj stavy a volej proceduru AlfaBeta s aktuálními parametry alfa a beta. Alfa nastavuj na maximální hodnotu z vráceného ohodnocení. Když vygenerujeme všechny možné tahy, vrátíme hodnotu alfa.
5. **Je-li na tahu protihráč:**
   Dokud je alfa < beta potom generuj stavy a volej proceduru AlfaBeta s aktuálními parametry alfa a beta. Beta nastavuj na minimální hodnotu z vráceného ohodnocení. Když vygenerujeme všechny možné tahy, vrátíme hodnotu beta.

Z obrázku níže je vidět, jak elegantně AlfaBeta funguje. Když například z jednoho uzlu vybere maximum 8 a dostane se do uzlu min, ten se expanduje a narazí na hodnotu 9, pak je jasné, že dál už generovat nemusí. I kdyby max. vybral hodnotu větší než 9, tak min vždy vybere při zpětném vrácení stav s hodnotou alespoň 8. Viz obrázek níže:


α = -∞,4,5
β = ∞
α = -∞
β = ∞,8,4
α = 4
β = ∞,5
α = 5
β = ∞,5
α = -∞,8 β = ∞
α = -∞,9 β = 8
α = -∞,2,4 β = 8
α = 4,5 β = ∞
α = 4,9 β = 5
α = 4,6 β = 5
α = 5 β = ∞
8 7 3 9 1 6 2 4 1 1 3 5 3 9 2 6 5 2 1 2 3 9 7 2 9 6 4

4 Návrh programu pro hraní šachů

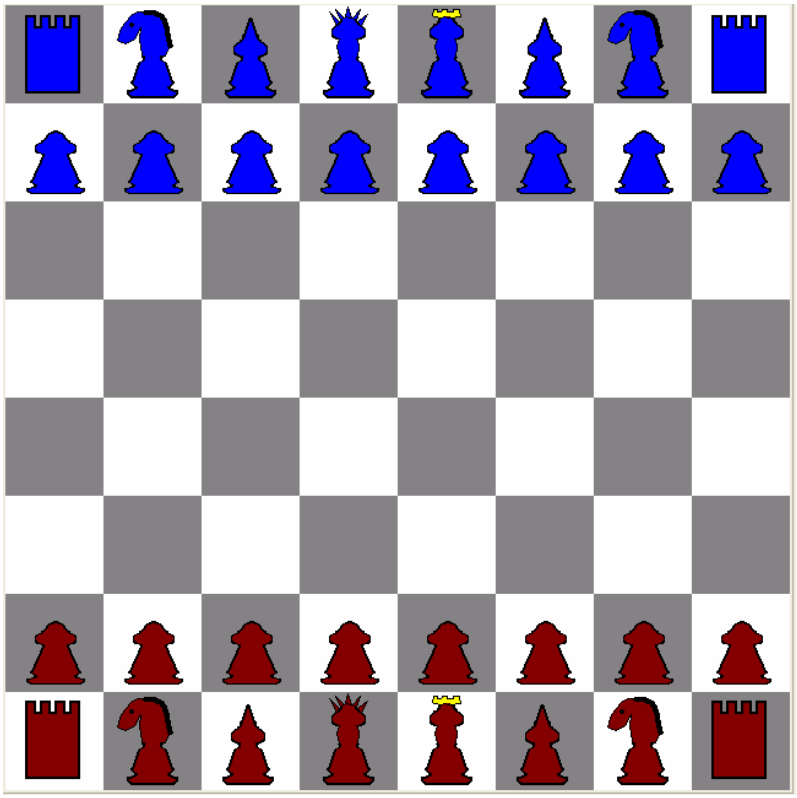
4.1 Úvod

Hru šachy asi není třeba nijak zvlášť představovat. Přesto zde uvedu základní shrnutí pravidel:

-Hraje se na čtvercové šachovnici 8*8, kde jsou střídavě tmavé a světlé plochy.
-Hráč má zpočátku k dispozici 16 figurek:

- 8 pěšáků
- 2 koně
- 2 střelce
- 2 věže
- 1 dámu
- 1 krále

Základní rozestavení je takovéto:



Úkolem hry je zajmout soupeřova krále. Hráči se postupně střídají v tazích. Každý táhne právě jednou zvolenou figurkou podle určitých pravidel. Pokud tah figurkou končí na místě jiné figurky, pak je tato

figurka vyřazena ze hry. Každá figurka může táhnout jinak.

Tahy se řídí podle následujících pravidel:

Pěšec

Pěšec může táhnout o jedno pole vpřed. Pokud s pěšcem nebylo dosud táhnuto, může se pohnout i o dvě políčka vpřed (za předpokladu, že na políčku před ním je volno).

Pokud chce pěšec zabrat soupeřovu figuru, musí tato figura být na levé nebo pravé diagonále před ním.

Pěšec se může proměnit v libovolnou figurku (kromě krále), pokud dorazí na protější konec šachovnice.

Střelec

Střelec se pohybuje po diagonále. Tudíž nikdy nemůže opustit tmavou nebo světlou plochu, která je daná pozicí na počátku hry.

Kůň

Kůň je jediná jednotka schopná přeskakovat cizí i své figurky. Pohybuje se o 1 políčko kterýmkoliv směrem a o 2 políčka směrem kolmým na původní směr pohybu.

Věž

Věž se pohybuje po šachovnici rovně, tedy ne po diagonálách. Tvoří tak doplněk střelce. Navíc s pomocí krále může provést **rošádu**. Ta bude uvedena níže.

Dáma

Dáma se umí pohybovat jako střelec a věž dohromady. Tedy si může vybrat, zda potáhne po diagonále, nebo přímo.

Král

Král se pohybuje o jedno políčko kolem sebe.

Král je hlavní figurka ve hře. Její ztráta znamená konec hry, tedy je jediná, která se nedá obětovat. Je-li král v ohrožení (další tah soupeře by vedl k zabrání krále), tato situace se nazývá **šach**.

Není-li možné krále nijak zachránit (tj. uhnout s ním, či představit jinou figurku, aby již král nebyl ohrožen) jedná se o **šach mat.** Hra v tomto stavu končí, jelikož by další tah vedl k zabrání krále.

Pokud hráč již nemůže táhnout žádnou figurkou kromě krále a tahem by se král dostal do šachu, jde o remízu nazvanou **šach pat**.

Rošáda

Rošáda je výměna pozic krále a věže. Tato výměna může proběhnout pouze tehdy, když:

- králem nebylo pohnuto
- král není v šachu a provedení rošády nezpůsobí šach
- dotyčnou věží nebylo pohnuto
- mezi věží a králem není žádná figurka

4.2 Návrh programu

Po pečlivé úvaze jsem zvolil princip založený na klasických metodách prohledávání stavového prostoru s využitím genetických algoritmů.

Genetické algoritmy pracují nezávisle na programu, takže je proberu později. Prohledávání stavového prostoru je založeno na technice minimaxu. Použil jsem i jiné, příbuzné metody, ale žádná neměla takový účinek jako jednoduchý minimax.

4.2.1 Použité datové struktury

Všechny struktury a funkce uvedené níže jsou součástí objektu CGame.

Uložení pozice figurek na hrací desce:

Pozice hry je uložena v poli 8*8, které může nabývat celočíselných hodnot. Každá figurka má přiřazenu určitou konstantu:

```
int Field[8][8];

#define PAWN   1          // pěšec
#define BISHOP 2          // střelec
#define KNIGHT 3          // kůň
#define ROOK   4          // věž
#define QUEEN  5          // dáma
#define KING   6          // král
```

Pokud prvek pole obsahuje hodnotu 0, jedná se o prázdné pole. Pokud obsahuje kladné hodnoty, jedná se o figurku hráče 1 a když záporné, jedná se o figurku hráče2.

Tedy hodnota 4 je věž hráče 1 a hodnota -4 je věž hráče 2.

Struktura tahu

V tazích se zaznamenává všechno potřebné pro provedení tahu a jeho vrácení. Také je zde zaznamenán stav hry, zda je nějaký král v šachu, zda může ještě provést rošádu apod.

```
typedef struct
{
   int OldX, OldY;
   int NewX, NewY;
   int Unit;
   int Type;
   int Pawn_Quest_Unit;
   int Move;
   int Status[2];
   int King_Can_Garrison[2];
   int King_Can_Garrison_Change[2];
   int First_State;
   int Closed;
   int Exist;
   int Score;
}TState;
```

OldX, OldY, NewX, NewY

Stará a nová pozice přesouvané jednotky.

Unit

Pokud tah způsobil zabrání figurky, potom nese její hodnotu, jinak je 0.

Type

Udává, zda se jedná o normální tah, rošádu, či proměnu pěšce v jinou jednotku.

Pawn_Quest_Unit

Pokud byl pěšec v tomto tahu proměněn v jinou figurku, nese tato proměnná její hodnotu.

Move

Který hráč provedl tah. +1 pro hráče 1 a -1 pro hráče 2.

Status[2]

Informace o šachu, šachu matu a šachu patu pro oba hráče.

```
#define STATE_CHECK 0                    // šach
#define STATE_CHECK_MATE 1               // šach mat
#define STATE_STALE_MATE 2               // šach pat
```

King_Can_Garrison[2]

Informace o tom, zda může hráč použít rošády. Informace je pro oba hráče.

King_Can_Garrison_Change[2]

Zda v tomto tahu došlo ke změně. Je to velmi důležitá informace pro vrácení tahu. Samotná informace o použití rošády nám nestačí. Při vrácení tahu je třeba vědět, zda tento tah způsobil změnu rošády a v případě, že byla předtím rošáda povolena a nyní není, nastaví se tato proměnná (pro příslušného hráče) na 1.

First_State

Na začátku hry nebyl proveden žádný tah, ale stav nese informace důležité pro další tahy - zejména o použitelnosti rošády. Pokud je First_State nastaven na 1, vrácení tahu zpět nemá žádný efekt. Tento tah nelze ani provést. Používá se jako startovní bod.

Closed

Pokud již nelze zahrát další tah (konec hry), je tento "flag" nastaven na 1.

Exist

Funkce na vrácení tahů potřebují vracet informaci o tom, zda už nelze z dané pozice žádný tah vygenerovat, použijí právě tuto proměnnou.

Score

Score vrácené minimaxem, nebo ohodnocovací funkcí.

4.2.2 Funkce pro generování tahů

```
// vrací pole, kam může pěšec, nebo co ohrožuje
TPos Pawn_Move(int x, int y, int* State);

// vrací pole, které pěšec ohrožuje
Pawn_Range(int x, int y, int* State);

// vrací pole, které král ohrožuje
TPos King_Range(int x, int y, int* State);

// vrací všechny tahy králem, včetně rošád
TPos King_Move(int x, int y, int* State, TState* S, int Player);

// vrací tahy věží
TPos Rook_Move(int x, int y, TPos* Pos, int *State);

// vrací tahy střelcem
TPos Bishop_Move(int x, int y, TPos* Pos, int *State);

// vrací tahy dámou
TPos Queen_Move(int x, int y, TPos* Pos, int *State);
// vrací tahy koněm
 TPos Knight_Move(int x, int y, int *State);
```

Všechny funkce vracejí strukturu TPos:

```
typedef struct
{
   int x, y;
   int Exist;
}TPos;
```

Exist původně udávalo, zda je tah platný, či ne. Ale později se ukázalo, že bude vhodné ho rošířit o informaci, zda se jedná o tah normální, rošádu či proměnu pěšce. Těmto hodnotám odpovídají konstanty:

```
#define POS_NONE 0
#define POS_NORMAL 1
#define POS_GARRISONING 2
#define POS_PAWN_QUEST 3
```

Původně si funkce udržovaly statickou proměnnou, která udávala, ve kterém stavu tahů jsou. Funkce se totiž volaly cyklicky a po každém zavolání vrátily další tah. Jenže to způsobovalo neznámé chyby. Proto jsem je nahradil odkazem na proměnnou, která nahrazuje stav. Po této změně byly chyby odstraněny.

Některé funkce mají koncovku Move a některé Range. Funkce s koncovkou Move se používají na generování tahů. Ty druhé při zjišťování ohrožování jednotek. Např. při zjišťování šachu nás nezajímá, zda pěšec může táhnout dopředu, ale zda krále ohrožuje.

4.2.3 Funkce pro provádění tahů a jejich kontrolu

void Move(TState *State);

Funkce provede stav State. Přesune figurku z pozice OldX, OldY na pozici NewX, NewY. Pokud tam byla jiná figurka, zapíše si ji do proměnné Unit. Je potřeba ošetřit situaci, kdy je tahem rošáda, nebo došlo k proměně pěšce.¨

void UndoMove(TState* State);

Tato funkce ruší účinky funkce Move.

int CanGarrisonLeft(int Player, TState* S);
int CanGarrisonRight(int Player, TState* S);

Podprogramy zjišťují, zda lze provést rošádu s levou, či pravou věží.

int IfKingIsInCheck(int Player);

Funkce najde na hrací ploše krále patřící hráči určenému dle vstupní proměnné Player. Pokouší se zjistit, zda je král v šachu. Dělá to tak, že testuje inverzně všechny figurky. Například vezme tahy koněm směřující od pozice krále a zjišťuje, jestli na pozicích určených tahy koněm není náhodou nepřátelský kůň. Je-li tam, potom je jasné, že krále ohrožuje. Stejně se to provede i se střelcem, věží, dámou, králem a pěšcem.

4.2.4 Funkce pro vstup a výstup

Jedná se o funkce uživatelského rozhranní, ukládání a nahrávání hry. Nebudu je zde rozebírat, protože pro samotný program nejsou nikterak zajímavé.

4.2.5 Ohodnocení stavů

Aby mohlo prohledávání stavového prostoru fungovat, je třeba mít něco, čím lze stavy ohodnotit a porovnat. O to se stará funkce:

TRes EvaluateGame(int Player, int UI_ID, int Move);

Player - který hráč ohodnocuje
UI_UD - index UI
Move - který hráč je na tahu

TRes je struktura:

```
typedef struct
{
   int Score;
   int Status[2];  // status hračů, CHECK, CHECK_MATE, STALE_MATE
}TRes;
```

Status hráčů je tam z historických důvodů, v novějších verzích je nahrazen funkcí pro kontrolu šachu, která se provádí po každém tahu ještě před ohodnocením.

Skóre je ohodnocení daného tahu.

Kritéria pro ohodnocení hry

Ohodnocovací vektor vektor

Ohodnocení je v programu zakódováno do jednorozměrného vektoru. Ten obsahuje samostatné proměnné i tabulky. Je to pro později využití genetických algoritmů.

Síla figurek

Každá figurka je ohodnocena určitou hodnotou (soupeřovy figurky jí mají zápornou). Toto kritérium je nejdůležitější a v průběhu hry by mohlo být téměř jediné.

Krytí figurek

Krytí figurek je tvořeno tabulkou, která pro každou dvojici figurek přiřadí body za krytí. Např. pokud pěšec kryje pěšce, dostane 5 bodů. Pokud kryje dámu, je to skoro nevýznamné, dostane 1 bod. Apod.

Ohrožování figurek v příštím tahu

Je podobné jako krytí, ale jedná se o ohrožování hráčem, který není na tahu.

Ohrožování figurek na tahu

Rozdíl mezi tímto ohrožováním a předchozím je následující. Pokud jsme na tahu a ohrožujeme střelcem soupeřovu dámu, dostaneme mnohem víc bodů, než kdybychom na tahu nebyli. Protože v našem případě je téměř jisté, že budeme střelce za dámu měnit. V druhém případě má dáma ještě šanci utéct a je to velice pravděpodobné.

Pozice pěšce

Pěšec, který stojí na políčku blíž nepříteli (tedy je velká šance, že v dalších tazích se promění v jinou figurku) by měl dostat za svojí snahu nějaké skóre. Také to podporuje v počátečních fázích hry lepší rozmístění jednotek.

Volná diagonála u střelce a volný sloupec (řádek) u věže

Za každé políčko, které střelec či věž ohrožuje je potřeba udělit taktéž nějaké body. Program potom staví tyto figurky do lepších pozic.

Volné místo kolem krále

Není dobré, když je král obestavěn kameny. Potom lze snadno docílit situace, kdy nemá při šachu kam uhnout. Za každé volné místo okolo musí dostat body.

Volná diagonála, řádek a sloupec u krále

Král by měl stát na takovém místě, aby mu nešlo dát snadno šach. Za každé políčko na diagonále, sloupci a řádku musí dostat nějaké záporné body.

Bílá a tmavá políčka

Pokud soupeř již nemá dámu a střelce (na bílé, či tmavé pozici), potom jednotky, které stojí právě na tmavém, či světlém políčku, dostanou kladné body. Tím ztíží soupeři napadání figur.

Šach, šach mat, šach pat

Skóre za šach pat je poněkud zvláštní. Může být kladné i záporné, podle toho, na co se rozhodne soupeř hrát. Pokud UI vyhodnotí (třeba poměrem skóre), že již nemá šanci normálně zvítězit, nastaví

si kladné skóre za šach pat. Potom se o něj bude snažit. V opačném případě za něj bude mít záporné skóre jako za šach mat a bude se mu snažit za každou cenu vyhnout.

4.2.6 Algoritmy pro generování tahů-minimax

Jedná se o nejjednodušší metodu a přesto velmi účinnou. Program postupně generuje tahy hráče na tahu. Z těchto tahů generuje tahy protihráče. Z nich zase další tahy hráče. Celé se to opakuje rekurzivně až do určitého zanoření, nebo do konce hry. Poslední vygenerované stavy se ohodnotí ohodnocující funkcí a postupně se vrací ty nejlepší tahy. Hráč na tahu si vybírá maximální skóre, protihráč zase minimální. Tato čísla se postupně dostávají až ke kořeni a na konci vidíme ten nejvýhodnější tah z celého stromu.

Vypadá to velice jednoduše, ale pro šachový program se to hůře implementuje. Je zde totiž pár faktorů, které implementaci ztěžují:

- šach, pokud jsme v šachu, nemůžeme táhnout jinak než aby šach v dalším tahu nebyl
- problém "exploze" tahů a následné až minutové zpracování jednoho tahu
- některé situace tím nelze zahrát - uvedu později

Minimax pro šachy:

Snaha o odhad počtu zanoření

Počet zanoření se bude během hry měnit. Zpočátku by neměl být příliš velký, jelikož ve hře je figurek mnoho a stejně nám velké zanoření k správnému rozehrání nepomůže. Je však těžké odhadnout předem jak velké zanoření má být. Je dobré spočítat hrací kameny a každému přiřadit určité body (za pěšce málo, za dámu hodně, protože dáma má mnohem víc možností tahů). Podle těchto bodů se rozhodnout do jaké hloubky se bude propočítávat.

Popis rekurzivní funkce minimaxu

Vstupy:

- tah, který vedl k tomuto stavu
- hráč, pro kterého se generuje strom
- hráč, který je na tahu
- zanoření

Výstup

- Vybraný tah

Funkce vrací nejlepší tah z dané pozice.

Nutno podotknout že stav celé hry (tedy rozmístění figurek na desce) je uložen globálně. Díky funkcím pro zahrání a vrácení tahu si stačí pamatovat pouze tahy, ne celý stav hry.

Tělo funkce

1. Inicializace

Pokud je na tahu hráč, pro kterého se strom generuje, nastav **MaxScore** na minimální možnou hodnotu (jakékoliv dostatečně vysoké záporné číslo). Pokud je na tahu protihráč, nastav **MaxScore** na maximální možnou hodnotu.

Zkontroluj počet zanoření a zvyš ho o 1. Počtem zanoření se myslí počet volání této funkce, který se udržuje v globální proměnné.

Pokud je příliš vysoký, pak již tento list neexpanduj. Předchozí list získá prázdný stav a bude sám sebe pokládat za konečný stav, kde dojde k ohodnocení. Toto pravidlo je dobré jako ochrana před "explozí stavů". Standardní počet zanoření je kolem 1000-3000. Ale někdy se může stát, že tento počet vyskočí třeba na 25000 a v dalším tahu klesne opět na 2000. Tuto situaci je těžké předvídat a propočet by trval moc dlouho. Proto je lepší ho dynamicky při výpočtu omezit.

Nastav příznak, že jsme zatím nenalezli žádný tah, který chceme vrátit.

2. Hlavní smyčka

- procházej celou šachovnici (x, y)

2.1 Pro každou figurku, kterou nalezneš

- nastav generování tahů na počáteční stav.

2.2 prováděj cyklus

- vygeneruj další tah
- pokud další tah už neexistuje, vyčerpali jsme je a vrať se do bodu 2 do hlavní smyčky
- pokus se ho zahrát
- nejde-li zahrát (král je po tomto tahu v šachu), vrať se na 2.2

- jestliže jsme v maximální hloubce:
    - zavolej funkci pro ohodnocení hry
    - zapamatuj si skóre

- jestliže nejsme v maximální hloubce:
    - zavolej rekurzivně funkci Minimax. Změň hráče na tahu na hráče opačného, zvyš počítadlo hloubky, vstupní tah
      je právě tento provedený tah
    - zapamatuj si skóre

- vyhodnocení skóre
    - pokud hraje hráč:
        - když je skóre tohoto stavu lepší, než dosavadní maximální skóre, zapamatuj si tento stav jako nejlepší
    - pokud hraje protihráč
        - když je skóre tohoto stavu lepší, než dosavadní maximální skóre, zapamatuj si tento stav jako nejlepší
- vrať tah zpět
- pokračuj v cyklu 2.2

3. ukončení

- nebyl-li nalezen žádný stav (tedy hráč již nemůže nikam táhnout), zkontroluj zda se jedná o šach mat, či o šach pat podle tohoto výsledku přiřaď skóre a vrať tento uměle vytvořený tah.
- v opačném případě vrať nejlepší nalezený stav

4.2.7 Výsledky experimentů pro minimax

Minimax přinesl velmi příznivé výsledky. Bez problémů porazí průměrné hráče. Mě osobně se ho ještě nepodařilo ani jednou přehrát. Testoval jsem to i na jiných hráčích a ti byli poraženi ještě snadněji, jelikož neznali algoritmus a jeho nedostatky. Proti šachovým přeborníkům jsem to zatím ještě nezkoušel, ale věřím, že kdybych si dal práci a program vylepšil, porazil by je taky.

Přesto minimax sám o sobě na některé problémy nestačí. Zejména na začátku hry je poněkud bezmocný. Nedokáže totiž hru rozehrát tak, aby získal později výhodu. Sleduje pouze několik tahů dopředu a chybí mu nějaká dlouhodobější strategie. Toto se velmi těžce implemetuje a je to doménou profesionálních programů. Tyto metody mají předem vytvořené herní tabulky na rozestavení určitých figurek a program se podle nich řídí. Vytvořit je by bylo nad mé časové možnosti, tudíž je můj program postrádá.

Pokud je hra rozehraná, ukáže se pravá síla minimaxu. Program již dokáže lehce přehrát nepříliš zkušeného hráče. Prohledáním stavového prostoru dokáže nalézt ty nejzákeřnější tahy, většinou v kombinaci s napadáním krále. V tomto stavu hry již lidský hráč téměř není schopen programu konkurovat, pokud nezíská převahu na začátku hry, nebo na něj nevymyslí nějakou šikovnou lest. Na to by ale musel znát herní algoritmus velmi dobře.

Na konci hry minimax může ovšem selhat. Zejména neumí dávat šach-mat v jasně vyhraných partiích. Pokud má k dispozici například dvě věže a soupeř má pouze krále, má s šach matem problém. Téměř všechny kombinace tahů totiž vedou k šach matu a jsou ohodnoceny stejným ohodnocením. V dalším tahu se stane totéž a program neví, který z tahů si vybrat. Může zahrát tah zpět, jelikož i ten by vedl po dalších tazích k jeho výhře. Jenže takto může uvíznout v nekonečné smyčce a je potřeba tento problém řešit jinak. Naštěstí se to nestává často, protože program většinou dává šach mat ještě v průběhu hry, když je ve hře ještě hodně figurek. Řešení tohoto problému v mém programu zatím není implementováno, ale pokusím se ho alespoň teoreticky navrhnout později v této dokumentaci.

4.2.8 Minimax s prořezáváním nevýhodných tahů

Jedná se o úpravu minimaxu. Použil jsem ji v první verzi mého šachového programu a v tomto již není implementovaná. Byla to zajímavá myšlenka, ale nakonec se neukázala příliš efektivní. Nicméně ji alespoň stručně popíšu.

Minimax má tu nevýhodu, že prohledává příliš mnoho stavů. Tento algoritmus se problém pokouší řešit. Funguje velmi podobně jako obyčejný minimax, ale v každém tahu se snaží předem vygenerovat všechny dostupné tahy z dané pozice, ty ohodnotí a vybere pouze určitý počet nejlepších

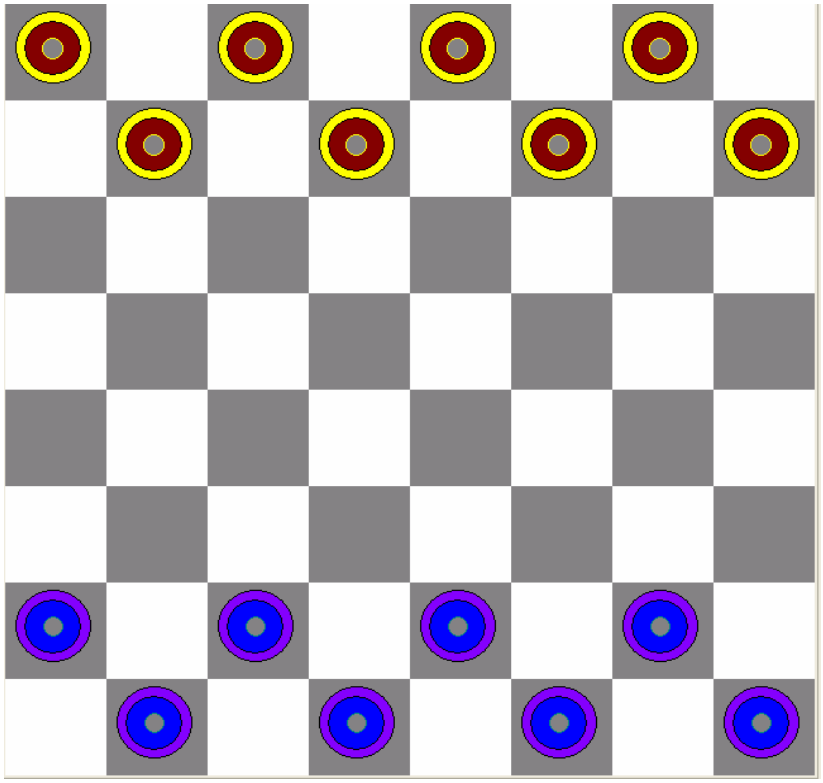
tahů. Tím se silně zredukuje počet prohledávaných tahů, v některých případech až 100 násobně! Ale brzy se ukázalo, že ty nejlepší tahy se zpočátku často jeví jako nevýhodné a jsou zahozeny. Navíc je potřeba v každém stavu volat hodnotící funkci, která vše zpomaluje. Tento algoritmus byl po vyzkoušení čistého minimaxu zavržen, neboť mu nemohl konkurovat. Ale nebyl zase tak špatný, hrál jako průměrný amatérský hráč šachu. Navíc prozkoumával více tahů dopředu a dalo by se říct, že se snažil o určitou strategii, tedy, že "přemýšlel" víc jako člověk narozdíl od klasického minimaxu.

5 Návrh programu pro hraní dámy

5.1 Úvod

Dáma je taktéž známá, uvedu základní shrnutí pravidel:

- hraje se pouze na černých políčkách šachovnice
- ve hře se vyskytují 2 typy figurek - obyčejní "pěšci" a dámy
- hráč začíná s 8 pěšci rozmístěnými na prvních dvou liniích šachovnice na černých políčkách



Pěšec

Pěšec smí táhnout buď vlevo nebo vpravo po diagonále, je-li cílové políčko volné. Pohyb smí být pouze dopředu - směrem k opačné části šachovnice vzhledem k hráči.

Dáma

Dáma se smí pohybovat po celé diagonále, ale nesmí při tom přeskakovat žádné figurky (vyjma braní, které bude popsáno za chvíli).

Braní pěšcem

Pokud je před pěšcem na diagonále cizí figurka, ale za ní (po té samé diagonále) je volno, potom může pěšec využít braní. To znamená, že se přesune o políčko za nepřátelskou figurkou a zabere jí (odstraní se ze hry).

Braní dámou

Dáma může brát podobně jako pěšec, ale nepřátelská figurka může být na diagonále libovolně daleko a dáma ji může "přeskočit" na libovolné pole za ní. Je zde ovšem jedno omezení. Při tomto braní nemůže přeskočit dvě figurky (nepřátelské ani vlastní).

Přednost při braní

Pokud je hráč v situaci, kdy může brát pěšcem i dámou, musí hrát dámou, jinak končí hru.

Nutnost braní

Pokud má hráč možnost brát, ale nevyužije to, končí hru.

Vícenásobné braní

Vícenásobné braní platí pro pěšce i dámu. Figurka, která dokončila braní a má možnost dalšího braní, může brát znovu. A tak dlouho, dokud je stále v situaci, že může brát.

Proměna pěšce

Pěšec, který dorazí na protější část šachovnice, se stává dámou. Výhody dámy však může využít až v dalším tahu. Toto pravidlo je důležité si uvědomit, protože kdyby neplatilo, tak by pěšec, který se stane dámou po braní, mohl použít pravidlo vícenásobného braní, ale nesmí. Je potřeba na to v programu pamatovat.

Nerozhodnost partie

Partie je nerozhodná, pokud během 15 tahů nedošlo k žádnému braní, či se stejná pozice vyskytla již třikrát, přičemž na tahu byl pokaždé stejný hráč.

5.2 Návrh programu

Struktura programu bude velice podobná jako ve hře šachy. Dáma je jednodušší než šachy, proto i implementace bude mnohem snazší. Ale je zde jeden problém, který se u šachů nevyskytuje. A to vícenásobné braní. Struktura tahu by potom musela odkazovat na nějaký seznam informací o sebraných figurkách. Jenže to by nestačilo. Během vícenásobného braní si může hráč často rozhodnout, jak bude braní pokračovat. Tedy by bylo potřeba nějak rekurzivně tyto tahy generovat a to by dalo moc práce. Mnohem jednodušší je jedno sebrání implementovat jako samostatný tah. Tím pádem bude klasický minimaxový strom vypadat trochu jinak. Nebudou se tam střídat tahy hráčů, ale některé tahy budou navazovat jako vícenásobné braní. Tato finta je velice jednoduchá a přitom tak snadno implementovatelná.

5.2.1 Použité datové struktury

Uložení pozice figurek na hrací desce:

Pozice hry je uložena v poli 8*8, které může nabývat celočíselných hodnot. Každá figurka má přiřazenu určitou konstantu:

```
int Field[8][8];

#define PAWN   1              // pěšec
#define QUEEN 2               // dáma
```

Pokud prvek pole obsahuje hodnotu 0, jedná se o prázdné pole. Pokud obsahuje kladné hodnoty, jedná se o figurku hráče 1 a když záporné, jedná se o figurku hráče2.

Tedy hodnota 2 je dáma hráče 1 a hodnota -2 je dáma hráče 2.

Na další stránce je uvedena struktura tahu:

Struktura tahu

V tazích se zaznamenává všechno potřebné pro provedení tahu a jeho vrácení.

```
typedef struct
{
        int OldX, OldY;
        int NewX, NewY;

        int Pawn_Quest;
..
        int JumpUnit;
        int JumpX, JumpY;

        int Player;
        int NextMove;

        int Score;
        int FirstState;
        int Exist;
        int PeaceTime;
}TState;
```

OldX, OldY, NewX, NewY

Stará a nová pozice přesouvané jednotky.

Pawn_Quest

Pokud je příznak nastaven na nenulovou hodnotu, znamená to, že se v tomto tahu pěšec proměnil na dámu.

JumpOver, JumpX, JumpY

JumpOver udává informaci o tom, zda byla přeskočena nějaká jednotka. Pokud je 0, nebyla přeskočena žádná. Jinak udává informaci, která jednotka to byla.

JumpX, JumpY potom udává pozici přeskočené jednotky.

Player, NextMove

Player udává hráče na tahu. NextMove udává, zda po tomto tahu následuje opět tah toho samého hráče - tento jednoduchý trik umožňuje vícenásobné braní. Příznak NextMove je důležitý hlavně pro to, aby se tyto tahy, vznikající vícenásobnými skoky, nezapisovali do paměti. O tom bude více později.

Score, FirstState, Exist

Score udává, jaké ohodnocení tento stav má. FirstState zase zda se jedná o kořenový stav (tedy neproveditelný, existující jen formálně) a Exist zda je tento stav platný.

PeaceTime

Určuje počet tahů, po které nebyla sebraná žádná figurka. Po 15 tazích se vygeneruje skóre za nerozhodnou partii, které je buď hodně velké, nebo hodně malé. Podle toho, jak se snaží UI partii zahrát. Když zjistí, že už má malou šanci na výhru, ohodnotí si remízu kladně, jinak záporně. Počítač se tak bude buďto snažit zahrát nerozhodnou partii, nebo se jí vyhnout.

Řešení uvíznutí ve hře

Řešení je téměř shodné s řešením popsaném ve hře šachy.

5.2.2 Funkce pro generování tahů

```
TState PawnMove(int X, int Y, int* State);
TState QeenMove(int X, int Y, TQueenState* State);
```

PawnMove

Funkce vrací tah pěšcem, či jedno jeho braní figurky. Vícenásobné braní se provede v dalším zřetězeném tahu.

QueenMove

Funkce vrací tah dámou.

5.2.3 Funkce pro provádění tahů a jejich kontrolu

void ExecuteState(TState *State);
void UndoState(TState*State);
int PlayAgain(int X, int Y);

ExecuteState, UndoState

Provede a vrátí tah.

PlayAgain

Tato funkce zjistí, zda figurka na pozicích X a Y může hrát znova. Tj zda může využít vícenásobné braní.

5.2.4 Ohodnocení stavů a kritéria pro ohodnocení hry

Kritéria jsou velice jednoduchá. Za obsazení zadních pozic (krytí před soupeřovou proměnou na dámu) jsou body. Za obsazení bočních pozic jsou také body (nedají se sebrat). Za krytí figurky se připisují body taktéž. Za možné braní kamenů se body nedávají, protože je bez generování dalších tahů obtížné to posoudit. Největší část skóre tvoří stejně ohodnocení figurek o zbytek se postará generování tahů.

5.2.5 Algoritmy pro generování tahů-minimax

Minimax pro dámu se liší od minimaxu pro šachy. Jsou zde dva hlavní důvody:

- Některé tahy mohou ovlivňovat tahy další. Jedná se o nutnost braní

a přednost braní dámou před pěšcem. Je potřeba tahy nejprve zjistit,
ořezat a potom teprve volat rekurzivně tahy další.

- Problém vícenásobných tahů. Řešení bylo již nastíněno výše. Pokud nastane situace, že figurka může pokračovat v braní, zavolá se rekurzivně funkce minimaxu, ale na tahu není protihráč, nýbrž stále stejný hráč. Dále se musí předat informace, že se nejedná o obyčejný tah, ale o vícenásobné braní. Tedy musí se předat informace o pozici figurky, která pokračuje v braní. K tomu je třeba ošetřit první volání minimaxu. Toto volání musí být v cyklu, který testuje, zda tento tah má být poslední, či se bude pokračovat. Je to proto, že minimax vrátí nejlepší tah a informaci, zda se jedná o vícenásobné braní, či ne. Poté se opět spustí minimax a generuje tahy tak dlouho, dokud braní nekončí. Nevýhoda tohoto principu, je že se minimax může volat vícekrát. Ale počet tahů bude dosti redukován, protože při vícenásobném tahu nevzniká příliš mnoho větvení a program má možnost prohlédnout si hru při každém generování do větší hloubky.

Minimax pro dámu:

Snaha o odhad počtu zanoření

Na rozdíl od šachů bude počet zanoření u dámy konstantní. Nedá se totiž dost dobře odhadnout, kdy dojde k vícenásobnému tahu, či proměně pěšce. Obě situace totiž dost značně ovlivňují počet tahů. Když máte méně figurek, dá se předpokládat, že počet tahů bude menší. Ale potom je větší riziko proměny pěšce, více místa na tahy a snižuje se počet vícenásobných tahů. Celkově se počet volání funkce během hry příliš nemění, narozdíl od šachů.

Popis rekurzivní funkce minimaxu

TState Minimax(TMinimaxInfo Info)

Vstupy:

- struktura TMinimaxInfo

Výstup

- Vybraný tah

Funkce vrací nejlepší tah z dané pozice.

Nutno podotknout že stav celé hry (tedy rozmístění figurek na desce) je uložen globálně. Díky funkcím pro zahrání a vrácení tahu si stačí pamatovat pouze tahy, ne celý stav hry.

Struktura TMinimaxInfo

```
typedef struct
{
   int Player;
   int Player_On_Move;
   int Deep;
   int MaxDeep;
   int FinalMove;
   int MoveX, MoveY;
}TMinimaxInfo;
```

Popis:

Player	-	Hráč, který se pokouší generovat strom tahů.
Player_On_Move	-	Hráč na tahu
Deep, MaxDeep	-	Dosažená a maximální hloubka tahů.
FinalMove	-	Final move udává, zda jde o vícenásobný tah.
MoveX, MoveY	-	Pozice figurky, která vícenásobný tah provádí.

Tělo funkce

1. Inicializace

Pokud je na tahu hráč, pro kterého se strom generuje, nastav **MaxScore** na nejnižší možnou hodnotu. Pokud je na tahu protihráč, nastav **MaxScore** na nejvyšší možné číslo.

Zkontroluj počet zanoření a zvyš ho o 1. Počtem zanoření se myslí počet volání této funkce, který se udržuje v globální proměnné.

Pokud je příliš vysoký, pak již tento list neexpanduj. Předchozí list získá prázdný stav a bude sám sebe pokládat za konečný stav, kde dojde k ohodnocení. Toto pravidlo je dobré jako

ochrana před "explozí stavů". Standardní počet zanoření je kolem 1000-3000. Ale někdy se může stát, že tento počet vyskočí třeba na 25000 a v dalším tahu klesne opět na 2000. Tuto situaci je těžké předvídat a propočet by trval moc dlouho. Proto je lepší ho dynamicky při výpočtu omezit.

Nastav příznak, že jsme zatím nenalezli žádný tah, který chceme vrátit.

Dále zkontroluj, zda nějaký pěšec nemá možnost braní. Pokud ano, zaznamenej si do proměnné braní hodnotu 1 (konstanta PAWN). Nyní toto proveď pro dámy. Pokud alespoň jedna dáma má možnost braní, zaznamenej do proměnné braní hodnotu 2 (konstanta QUEEN). Tato kontrola se provede pouze tehdy, když se nejedná o vícenásobný tah.

2. Hlavní smyčka

- procházej celou šachovnici (x, y)

Pokud se jedná o vícenásobné braní, nastav x a y na pozici předanou v MinimaxInfo na MoveX a moveY.

2.1 Pro každou figurku, kterou nalezneš

- nastav generování tahů na počáteční stav.

2.2 prováděj cyklus

- vygeneruj další tah
- pokud se nejedná o vícenásobný tah a tah neodpovídá pravidlům(např. táhneme pěšcem a máme brát dámou), vracíme se do hlavní smyčky.

- pokud další tah už neexistuje, vyčerpali jsme je a vrať se do bodu 2 do hlavní smyčky
- pokus se ho zahrát

- jestliže jsme v maximální hloubce:
    - zavolej funkci pro ohodnocení hry
    - zapamatuj si skóre

- jestliže nejsme v maximální hloubce a nelze zahrát vícenásobný tah:
    - zavolej rekurzivně funkci Minimax. Změň hráče na tahu na hráče opačného, zvyš počítadlo hloubky
    - zapamatuj si skóre

- jestliže nejsme v maximální hloubce a lze zahrát vícenásobný tah:
    - Předej tuto informaci a zavolej rekurzivně funkci minimax. Zapamatuj si skóre.

- vyhodnocení skóre
    - pokud hraje hráč:
        - když je skóre tohoto stavu lepší, než dosavadní maximální skore, zapamatuj si tento stav jako nejlepší
    - pokud hraje protihráč
        - když je skóre tohoto stavu lepší, než dosavadní maximální skore, zapamatuj si tento stav jako nejlepší
- vrať tah zpět
- pokračuj v cyklu 2.2

3. ukončení

- nebyl-li nalezen žádný stav (tedy hráč již nemůže nikam táhnout), zkontroluj zda se jedná o výhru, prohru či remízu.
  podle tohoto výsledku přiřaď skóre a vrať tento uměle vytvořený tah.
- v opačném případě vrať nejlepší nalezený stav

5.2.6 Výsledky experimentů pro minimax

Experimenty přinesly podobný výsledek jako u šachů. Počítač je takřka neporazitelný, ale jak to už dámy bývá, není takový problém uhrát remízu.
Ukázalo se také, že nemá příliš cenu vymýšlet nějaké složité hodnotící funkce. Program hrál nejlépe pouze tehdy, když se bralo v potaz pouze ohodnocení figur. V dámě totiž není oproti šachům takový důraz na strategii, stačí pouze prohledat pár tahů dopředu. A v tom je počítač bez konkurence.

6 Genetické algoritmy

Genetické algoritmy vychází z principů přirozeného výběru - evoluce. Jedinci, kteří se lépe adaptují na prostředí, mají větší šanci na přežití a větší šanci na plození potomků. Tito potomci mají podobné vlastnosti jako jejich rodiče a jsou také schopni se adaptovat na prostředí.

Genetický algoritmus vypadá takto:

1. **Start** – tvorba počáteční populace
2. **Ohodnocení populace**
3. **Testování –** splňuje-li populace zadané podmínky, můžeme algoritmus ukončit
4. **Výběr rodičů –** zde vybereme několik jedinců, kteří v ohodnocení dopadli nejlépe
5. **Rekombinace –** tvorba potomků
6. **Mutace potomků**
7. **Tvorba nové populace**
8. **Zpět na ohodnocení**

Je vidět, že se algoritmus příliš neliší od evoluční teorie. Na počátku se vygeneruje náhodná populace. U ní se předpokládá, že není schopna splnit podmínky, na ní kladené. Populace se ohodnotí, podle kritérií daných podmínkami. Pokud alespoň jeden jedinec splňuje podmínky, algoritmus dosáhnul dobrého výsledku a můžeme ho ukončit. Algoritmus můžeme ukončit také pokud žádný z jedinců po delší dobu podmínky nesplňuje. Pokud nesplňuje podmínky, vybereme několik nejlepších jedinců (kteří zadané podmínky splnili nejlépe). Hodnocení jedinců probíhá na základě vyhodnocení, jak moc se liší řešení představované tímto jedincem od řešení, které požadujeme.

Výběr rodičů je podstatnou složkou genetického algoritmu. Požadujeme, aby lepší jedinci měli větší šanci na plození potomků a tím pádem se populace mohla zlepšovat. Existuje na to několik algoritmů:

Výběr rodičů

Algoritmus výběru elity

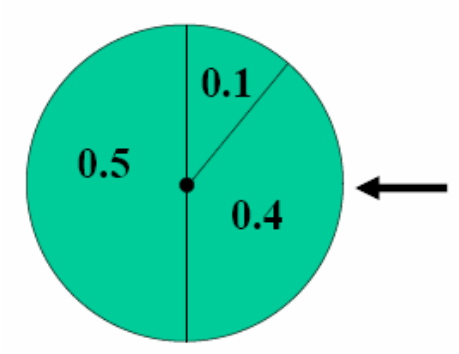
Jednoduše se jedinci seřadí podle jejich ohodnocení a potom se vybere určitý počet nejlépe ohodnocených jedinců – obvykle kolem 20%.

Algoritmus turnaje

Z populace je náhodně vybrán předem daný počet jedinců a nejlepší jedinec z této skupiny je vybrán jako rodič. Tuto operaci provedeme tolikrát, kolik potřebujeme rodičů.

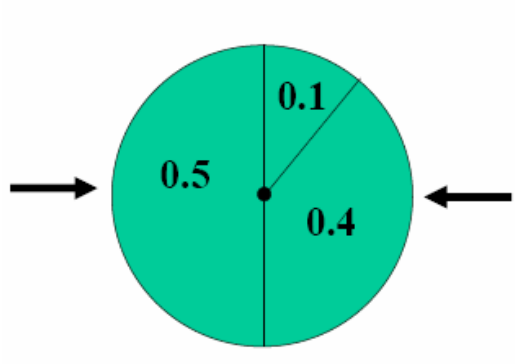
Algoritmus rulety:

Výseče kola rulety jsou přímo úměrné přepočteným hodnotám úspěšnosti pro jednotlivé jedince. Po zastavení kola je vybrán jedinec, na který ukazuje jazýček rulety. Kolo se musí roztočit tolikrát, kolik rodičů musí být vybráno. Tento způsob zajišťuje, že lepší jedinci mají větší šanci stát se rodiči, protože zabírají větší plochu na ruletovém kolu.



Algoritmus stochastického univerzálního vzorkování:

Výseče kola jsou opět úměrné přepočteným hodnotám úspěšnosti pro jednotlivé jedince. Po zastavení kola jsou vybráni jako rodiče jedinci, na jejichž výseče ukazují rovnoměrně rozložené jazýčky, kterých musí být tolik, kolik rodičů má být vybráno. Kolo se roztáčí pouze jednou.



Rekombinace

Rekombinace vytváří nové jedince z dříve vybraných rodičů. Jednou z metod rekombinace je metoda křížení.

Jednobodové křížení

Stanoví se náhodně místo křížení a dítě získá část genů prvního rodiče (dle bodu křížení) a část genů druhého rodiče.

Příklad:

Geny prvního rodiče: abcdef
Geny druhého rodiče: 123456

Bod křížení se náhodně zvolí na pozici za 3. genem. Tedy nový potomek bude mít geny:

Rodič1 abcdef
Rodič2 123456
Potomek1 abc456
Potomek2 123def

Vícebodové křížení

Princip je stejný jako u jednobodového křížení, jen se náhodně vybere více bodů křížení. Děti vzniknou prohozením vybraných částí obou rodičů.

Uniformní křížení

Každý gen se přenese s pravděpodobností 0.5 od prvního rodiče, nebo od druhého rodiče.

Mutace

Je určitá šance, že nový jedinec podlehne mutaci, tj. náhodné změně genů nezávislé na rodičích. Vybere se náhodně určitý počet jedinců (například 1% populace) a u těch se náhodně změní hodnota genu. U binárních genů se invertuje, u reálných hodnot se přičte nějaká relativně malá náhodná reálná hodnota (většinou 0-10%).

Zařazení potomků do nové populace

Mohou nastat různé případy

- počet potomků je stejný jako počet rodičů. Tito jedinci plně nahradí ty předchozí.
- Počet jedinců je menší než počet rodičů a noví jedinci nahradí nejslabší z předchozí populace – nejhůře ohodnocené.
- Počet jedinců je menší než počet rodičů a noví jedinci nahradí náhodně vybrané rodiče.
- Počet potomků je větší než počet rodičů. Do nové populace se zařadí potomci s nejlepším ohodnocením.

Využití v šachovém programu

Nejprve se zmíním, proč genetické algoritmy nejsou vhodné pro dámu. Jak jsem již řekl, u dámy není třeba složitě počítat ohodnocení stavů a genetický algoritmus by v tomto případě UI příliš nezlepšil. Ale u šachů je situace jiná.

V návrhu šachového programu jsem popisoval, že parametry ohodnocení jsou zakódovány do jednorozměrného pole obsahující reálné prvky. To proto, aby se ně daly aplikovat genetické algoritmy. Princip je jednoduchý a vychází z teorie nastíněné výše. Vektor ohodnocení v podstatě obsahuje genetickou informaci. Náhodně se vygeneruje určitý počet jedinců, náhodně vybrané dvojice jedinců spolu sehrají partie. Ohodnotí se podle dosažených výsledků. Potom se na ně použije výběr rodičů a rekombinace s mutací. Vznikne nová populace a algoritmus se opakuje. Samozřejmě zde nemohou být žádná vnější kritéria, protože ty jsou až finální hra proti člověku. Dá se ovšem předpokládat, že jedinec, který poráží ostatní jedince, bude mít i větší šanci proti člověku.

Bohužel algoritmus v podobě, který jsem zde popsal, není implementován a vyzkoušet. Byl by velice náročný a abych stihl výsledky dodat do obhajoby, potřeboval bych superpočítač. Proto jsem genetický algoritmus použil v jednodušší podobě. Sám jsem si vytvořil několik jedinců a spustil automatickou hru. Když hra skončila, vybral jsem vítěze a spustil další duel. Z těchto vítězů jsem

jednobodovým křížením vygeneroval nové jedince. A zkusil si proti nejlepšímu zahrát. Skutečně hrál lépe, než slabší jedinci z předchozích generací. Kdybych měl dostatečně silný výpočetní stroj a čas, určitě by genetické algoritmy dosáhly dobrých výsledků.

7 Neuronové sítě

Aplikace neuronových sítí v šachovém programu a programu pro dámu je již obtížnější než genetické algoritmy. Po nastudování vícevrstvé perceptronové sítě mě napadla možnost použít tuto síť jako klasifikátor, který ohodnocuje daný stav hry a na svých výstupech „říká“ jak moc dobrá situace ještě je.

U dámy je tato aplikace naprosto zbytečná, jak jsem již několikrát říkal, dáma hraje nejlépe s jednoduchým minimaxem bez složitých ohodnocení.

U Šachů by byla zase potřeba obrovská síť s velkým počtem vstupních neuronů. Muselo by jich být opravdu hodně, nejlépe několik neuronů na každou pozici na šachovnici. Tyto neurony by snímaly každý určitou figurku pro každého hráče – tj. ve hře je 6 figurek, 2 hráči – tj. 12 neuronů na políčko. Šachovnice obsahuje 64 ploch, to znamená, že počet neuronů ve vstupní vrstvě by byl 12 * 64. Vezměme-li v úvahu, že tato síť musí mít alespoň jednu skrytou vrstvu a v literatuře se doporučuje dvojnásobný počet neuronů, je tato síť neúnosně velká a bylo by velmi obtížné ji natrénovat.

Další problém je nízká reakce na drobné odchylky na vstupech. Představte si situaci, že král stojí na určité ploše a hrozí mu šach mat. A druhou situaci, kdy stojí vedle a nehrozí mu žádné nebezpečí. Dá se předpokládat, že síť tyto dvě naprosto odlišné situace vyhodnotí podobně. Po aplikaci minimaxu (kdy se síť použije na ohodnocení stavů) by nám vyšel naprosto nesmyslný tah. Z těchto důvodů jsem se nepokoušel neuronové sítě pro hraní šachů ani implementovat.

8 Závěr

Metody soft computing v porovnání s klasickými metodami příliš neobstály. Programy pro hraní dámy a šachů se neukázaly vhodným příkladem. Zde klasické metody naprosto zvítězily. Metody soft computing šly rozumně použít jedině v kombinaci s klasickými přístupy.

Program pro hraní šachů s minimaxem bez problémů porazí průměrného hráče šachu. Jeho síla je především ve střední fázi hry, kdy nalézá velmi dobré tahy. Problém je v počátku, kdy „nevidí“ příliš dopředu a tudíž má problém rozehrát hru tak, aby z rozestavení mohl těžit.

Dáma se ukázala naprosto nevhodná pro jakékoliv experimenty. Protože nakonec hrála nejlépe, když ohodnocení tahu probíhalo na základě hodnoty hracích kamenů. Dáma nemá tolik možností tahů a tedy lze dosáhnout vyššího zanoření než u šachů.

A jaký to pro mě mělo přínos? Především jsem se naučil, že dobrý návrh urychlí implementaci a výrazně sníží riziko chyb. Rád bych se věnoval problematice UI v počítačových hrách i nadále v diplomové práci.

Literatura

[1] Mařík, Štěpánová, Lažanský a kolektiv, Umělá inteligence 1-4. Praha, Akademia 1993.

[2] Aleš Kubík, Inteligentní agenty. Brno, Computer Press 2004.

[3] Doc. Ing František Zbořil CSc., Ing. Petr Hanáček, Umělá inteligence, Brno, Ediční středisko VUT Brno 1990

[4] Doc. Ing František Zbořil CSc., skripta do předmětu Soft computing

Seznam příloh