Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE

Kristián Kacz

Porovnání šachových strategií

Katedra Aplikované Matematiky

Vedoucí bakalářské práce: RNDr. Josef Cibulka
Studijní program: Informatika, obecná informatika

2010

V první řadě bych se chtěl poděkovat RNDr. Josefu Cibulkovi za vedení této práce, za cenné připomínky a trpělivost. Také bych se rád poděkoval rodině za podporu a korektúru.

Prohlašuji, že jsem svou bakalářskou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce a jejím zveřejňováním.

V Praze dne 21.05.2010 Kristián Kacz

Obsah

Název práce: Porovnání šachových strategií
Autor: Kristián Kacz
Katedra (ústav): Katedra Aplikované Matematiky
Vedoucí bakalářské práce: RNDr. Josef Cibulka
e-mail vedoucího: cibulka@kam.mff.cuni.cz

Abstrakt: Cieľom práce je poskytnutie prehľadu prístupov v počítačových šachoch. Navrhuje a implementuje šachový engine pre viachráčový sieťový šachový program ChessNet. V rámci enginu implementuje niekoľko známych prehľadávacích algoritmov ako Negamax, Alpha-beta a Negascout a upozorňuje na ich slabé body. Prostredie ChessNet je doplneno možnosťou pre porovnanie šachových enginov. Naimplementované algoritmy porovnáva z pohľadu časovej zložitosti. Ukazuje niekoľko faktorov, ktoré treba brať do úvahy pri vytvorení funkcie na ohodnotenie stavov. Niekoľko takých funkcií porovnáva z pohľadu úspešnosti voči sebe.

Klíčová slova: šach, AI, minmax, alpha-beta, negascout

Title: Comparing chess strategies
Author: Kristián Kacz
Department: Department of Applied Mathematics
Supervisor: RNDr. Josef Cibulka
Supervisor's e-mail address: cibulka@kam.mff.cuni.cz

Abstract: The aim of this work is to provide an overview of approaches in computer chess. It designs and implements a chess engine for multiplayer network chess program ChessNet. Within the engine implements several search algorithms like Negamax, Alpha-beta, Negascout and points to their weaknesses. Adds a possibility to the ChessNet environment to compare chess engines. Compares implemented algorithms in terms of time complexity. Shows several factors wich we have to take into consideration during the design of evaluation function of game states. Implements some such functions and compares them in terms of success against each other.

Keywords: chess, AI, minimax,alpha-beta, negascout

Kapitola 1

Úvod

Šach je jednou z najznámejších logických hier. Jeho prvá forma pochádza zo 6. storočia z Indie. Počas storočí sa rozšíril po celom svete, postupne sa menil, pridali sa nové pravidlá, vytvorila sa kopa rôznych variantov. Vždy bol považovaný za vhodný spôsob súťaženia logického myšlenia. Dnes je naňho postavená profesionálna športová disciplína.
V človeku popri tom, že sa chcel naučiť čo najlepšie hrať, sa ukrývala aj túžba vytvorenia stroja hrajúceho šach. Najznámejší pokus patrí Wolfgangovi Kempelenovi, ktorý sa v 19. storočí preslávnil so svojím šachovým strojom Turek [2]. Neskôr sa o ňom dozvedelo, že to nebol ozajstný automat, ale skrýval v sebe šachistu. Naozajstné šachové stroje sa vytvorili až po prvých počítačoch. Prvý odborný článok zaoberajúci sa s touto problematikou vyšiel v roku 1950 [10]. Necelých 50 rokov muselo uplynúť, kým prvýkrát počítač vyhral zápas nad svetovým šampiónom: v roku 1996 program Deep Blue porazil Garryho Kasparova v prvej hre turnaja [3].

1.1 Štruktúra bakalárskej práce

V kapitole 2 si ukážeme začiatok histórie počítačových šachov. Zavedieme základný problém a ukážeme možné spôsoby riešenia. V sekcii 2.1 opíšeme základný algoritmus, riešiaci náš problém. V sekciách 2.2 až 2.5 ukážeme, ako môžeme predchádzajúci algoritmus zefektívňovať. V sekcii 2.6 popíšeme jeden slabý bod tohto algoritmu a ukážeme riešenie. Sekcia 2.7 poukazuje na faktory, ktoré treba brať do úvahy pri konštrukcii ohodnocovacej funkcie stavov. Kapitola 3 obsahuje popis podobných projektov. Nakoniec kapitoly 4 a 5 obsahujú programátorskú a užívateľskú dokumentáciu projektu.

Kapitola 2

Teória počítačového šachu

Z pohľadu teórie hier, šach je deterministická hra (náhoda nehrá rolu) s úplnou informáciou a s nulovým súčtom (tzv. *zero-sum game* - súčet ziskov a strát všetkých hráčov je rovný nule). Vďaka týmto vlastnostiam pre každý stav hry platí jeden z nasledujúcich tvrdení [7]:

- je výherný pre bieleho hráča (biely má výhernú stratégiu)
- je remízový (oba hráči môžu mať stratégiu, ktorá vedie k remíze)
- je výherný pre čierneho hráča (čierny má výhernú stratégiu).

Keď by sme vedeli určiť o danom stave hry, že do ktorej kategórie patrí, hráči by mohli postupovať podľa optimálnej stratégie, a každá hra by skončila rovnakým výsledkom. K tomu by sme potrebovali postaviť *strom hry* - strom možných postupov hry, v ktorom uzly sú stavy hry, synovia uzla sú stavy, ktoré môžu vzniknúť jedným legálnym ťahom zo stavu reprezentovaného pôvodným uzlom, listy sú koncové stavy. O koncových stavoch vieme, že do ktorej kategórie patria a ostatné stavy spiatočne kategorizujeme podľa toho, či hráč na ťahu je schopný vykonať ťah, ktorý vedie do stavu, kde on vyhráva, alebo aspoň remizuje. Priemerný počet možných ťahov v danom stave je 30 a priemerná dĺžka jedného zápasu je 40 ťahov pre oboch hráčov. Z toho dostaneme odhad $(30 * 30)^{40} \approx 10^{120}$ na počet možných postupov hry [10]. Skonštruovať taký strom je v súčasnosti prakticky nemožné, preto dokonalú stratégiu pre šach nie je možné nájsť. Vďaka tomuto faktu sa dá táto oblasť stále rozvíjať.
Cieľom je nájsť čo najlepšiu stratégiu, teda algoritmus, ktorý v danom stave hry vyberie ťah, ktorý vedie smerom k výhre. Shannon v článku [10] sformuloval dva možné prístupy k tomuto problému:

- stratégia „typu A“ je založená na tzv. ohodnocovacej funkcii, ktorá pre daný stav hry vráti číslo. Čím je väčšie toto číslo, tým je stav lepší pre prvého hráča a horší pre druhého. Algoritmus postaví strom hry do určitej hĺbky, uzly na tejto úrovni ohodnotí. Prvý hráč chce vybrať ťah smerujúci k najväčšej hodnote, druhý hráč k najmenšej. Takto spiatočne z listov sa ohodnotia všetky uzly a algoritmus nakoniec vyberie ťah.

- stratégia „typu B“ nepostaví úplný strom všetkých možných ťahov, ale berie do úvahy len „silné“ ťahy, také, ktoré oveľa zmenia ohodnotenie stavu (branie figúrky, promócia). Týmto dostane zriedkavejší strom, a s rovnakou výpočtovou silou môže prehľadať do väčšej hĺbky.

Väčšina úspešnych programov je založená na niektorom z týchto algoritmov. Algoritmy typu A majú veľkú výpočtovú potrebu a preto môžu prehľadávať strom len do veľmi malej hĺbky, ale zaiste vrátia ťah smerujúci k najlepšiemu možnému výsledku v tej hĺbke. Algoritmy typu B naopak majú menšie potreby, prehľadávajú do väčšej hĺbky, ale ich efektivita závisí od výberu prehľadávaných ťahov. Jeden z prvých zápasov v počítačových šachoch sa odohrával medzi implementáciami týchto typov: v roku 1966 program Kotok-McCarthy typu B zo Stanfordskej univerzity hral štyri hry proti sovietskemu programu z Institute for Theoretical and Experimental Physics, ktorý bol typu A. Ruský program zvíťazil 3:1, a od tej doby sa algoritmy typu A považovali za správny smer.
Existujú samozrejme iné prístupy, ktoré sú buď optimalizáciami predchádzajúcich algoritmov (alpha-beta a iné orezávania, opening- a endgame-tabuľky), alebo celkom odlišné experimenty (neurónové siete).

2.1 Minimax

Základný princíp stratégií typu A je úplne prehľadávanie stromu hry do danej hĺbky. Hráči ťahajú striedavo a majú opačné ciele - prvý hráč chce smerovať k vysoko ohodnotenému stavu, druhý k nízkemu. Algoritmus minimax simuluje ich výbery: prehľadáva strom do hĺbky, na poslednej vrstve zavolá ohodnocovaciu funkciu, získané hodnoty propaguje nahore vždy vybraním minima alebo maxima z nich.
Algoritmus sa najčastejšie používa vo forme *Negamax*, ktorý berie do úvahy fakt $max\,(a_i) = -min\,(-a_i)$. Formálny popis algoritmu:

Negamax(*stav*, *hĺbka*):integer
if (*stav* je koncový **or** *hĺbka* = 0) **then**
  return eval(*stav*)
end if
result := -∞
for all potomok *stavu* **do**
  result := max(*result*, -**Negamax**(*potomok*, *hĺbka*-1))
end for
return *result*

Algoritmus rekurzívne volá sám seba, preto jeho zložitosť môžeme merať počtom volaní ohodnocovacej funkcie na poslednej vrstve. Tento počet je $O(b^d)$, kde b je priemerný počet možných ťahov a d je hĺbka prehľadávania. Tomu odpovedajú aj namerané hodnoty počtu volaní ohodnocovacej funkcie v listoch, uvedené v tabuľke 2.1 ktoré som dostal ako priemer cez 1000 volaní programu `negamax.exe` s určitou hĺbkou. Priemerný počet možných ťahov dostaneme ako príslušnú odmocninu z nameraných hodnôt. Táto hodnota vyšla menej ako 30 (priemerný počet spomenutý v úvode tejto kapitoly), čo môže byť odôvodnený tým, že opísanému algoritmu na konci hry môže dlho trvať, kým nájde cestu k matu. V týchto situáciách aspoň jeden hráč má málo možných ťahov. Hodnoty som meral cez celé zápasy hrané týmto algoritmom, a tak situácie s malým počtom možných ťahov dostali väčšiu váhu v priemere ako majú naozaj. Napriek tomu vidíme, že beh algoritmu exponenciálne predĺžuje zväčšením hĺbky prehľadávania. Ak chceme dosiahnuť lepšie výsledky, musíme prehľadávanie zrýchliť.

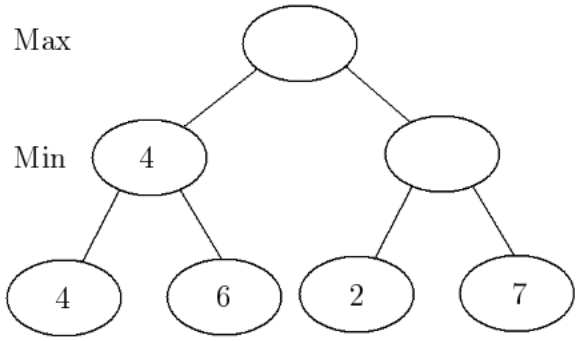
hĺbka	# listov	# možných ťahov
1	20,6	20,6
2	424	20,6
3	10965	22,2
4	266756	22,7
5	*n/a*	*n/a*

Tabuľka 2.1: Negamax

2.2 Alpha-beta orezávanie

Pri prehľadávaní stromu minimaxovým algoritmom sa často stane, že sa prehľadávajú podstromy, ktoré nemajú žiadny vplyv na výsledok. Uvažujme situáciu z obrázku 2.1 . Maximalizujúci hráč na prvej vrstve po prehľadávaní prvého podstromu vie, že výsledná hodnota bude aspoň 4. Hodnota druhého podstromu ale nemôže byť väčšia ako 2. Preto ostatné jeho podstromy nemusíme prehľadávať, lebo ich výsledok neovplyvňuje hodnotu na prvej vrstve. Vynechaním skúmania týchto podstromov by sme boli schopní zjavne zrýchliť beh algoritmu. K tomu potrebujeme si zapamätať interval (α, β) vyhovujúcich výsledkov. Keď maximalizujúci hráč nájde podstrom s hodnotou vyššou ako β, alebo minimalizujúci hráč nájde podstrom s hodnotou menšou ako α, ostatné podstromy môžeme orezať.
Formálny popis algoritmu:

Alphabeta(*stav*, *hĺbka*, α, β):integer
if (*hĺbka* = 0 **or** *stav* je koncový) **then**
  return eval(*stav*)
end if
for all potomok *stavu* **do**
  α := max(α, -**Alphabeta**(*potomok*, *hĺbka*-1, -β, -α))
  if β=α **then**
    break
  end if
end for
return α



Obr. 2.1: Alpha-beta

Pomer zrýchlenia závisí na poradí preskúmaných ťahov. V najhoršom prípade nedôjde k žiadnemu orezávaniu a beh algoritmu zostane $O\left(b^d\right)$. Ak by sme mali optimálne zoradenie ťahov, teda medzi všetkými možnými ťahmi by sa ten najlepší nachádzal na prvom mieste, mohli by sme dosiahnuť maximálne zrýchlenie. Knuth v článku [5] rozdelil vrcholy stromu hry do troch kategórií:

- vrchol typu 1 (tzv. PV-node) je ten, ktorého hodnota padne do intervalu (α, β). Prehľadávajú sa všetky jeho podstromy. Koreň stromu hry a prvé vrcholy z ľava na všetkých vrstvách sú typu 1.
- vrchol typu 2 (tzv. cut-node) je ten, ktorého hodnota je aspoň β. Má aspoň jedného potomka typu 3. Vždy sa prehľadáva aspoň jeden jeho podstrom. Neohodnotí sa presnou minimaxovou hodnotou, ale dolnou hranicou tej hodnoty.
- vrchol typu 3 (tzv. all-node) je ten, ktorého hodnota je menšia alebo rovná α. Všetky jeho potomkovia sú typu 2. Prehľadávajú sa všetky jeho podstromy. Dostane hodnotu hornej hranice jeho presnej minimaxovej hodnoty.

Označme PV(d), cut(d), all(d) počet listov v podstromoch hĺbky d s koreňom typu 1, 2 a 3. Platí: PV(0) = cut(0) = all(0) = 1. Pri optimálnom zoradení prvý možný vykonaný ťah vedie k najvyššej hodnote. Vo vrcholoch typu 1 tento ťah zvyšuje hodnotu α a všetky ostatné vrcholy budu typu 2. Vo vrcholoch typu 2 tento ťah vedie do vrcholu typu 3 a ostatné podstromy sa neprehľadávajú. Podľa tohto pozorovania pre d>1 platí:

$$cut(d) = all(d-1)$$

$$all(d) = b * cut(d-1)$$

$$PV(d) = PV(d-1) + (b-1) * cut(d-1)$$

Z toho môžeme odvodiť explicitné vzťahy:

$$cut(d) = b^{\lfloor d/2 \rfloor}$$

$$all(d) = b^{\lceil d/2 \rceil}$$

$$PV(d) = b^{\lfloor d/2 \rfloor} + b^{\lceil d/2 \rceil} - 1$$

Koreň stromu hry je typu 1, preto dolný odhad zložitosti algoritmu bude $\Omega\left(b^{\frac{d}{2}}\right)$. Algoritmus s takou zložitosťou by mohol prehľadávať strom do dvojnásobnej hĺbky voči Minimaxu. Optimálne zoradenie ťahov je však rovnako ťažká úloha: keby sme vedeli určiť optimálne poradie, nemuseli by sme strom vôbec prehľadávať, stačilo by vždy ťahať ten prvý ťah. Namiesto optimálneho poradia nám bude stačiť zoradenie, ktoré opíšeme v sekcii 2.4.
V tabuľke 2.2 z nameraných hodnôt z programu `negaalphabeta.exe` aj bez použitia špeciálneho zoradenia ťahov vidíme dôležitosť tejto idey. Tieto hodnoty sú ešte veľmi ďaleko od teoretického minima, ale pridaním zoradenia sa k nemu postupne priblížime.

hĺbka	# listov
1	20,6
2	121
3	1699
4	12954
5	154425

Tabuľka 2.2: Alpha-beta

2.3 Transpozičná tabuľka

Pri prehľadávaní stromu hry sa občas stane, že sa s tým istým stavom hry stretneme viackrát. Uvažujme len prípad, keď prvý hráč vyberie ťah A, súper odpovedá ťahom B a prvý nakoniec zvolí C. Keď legálnosť vybraných ťahov sa nemenila počas tejto hry, s ťahmi C, B, A by sme sa dostali do tej istej situácie. Takéto stavy nazývame transpozíciami. Keď sa najlepšia hodnota propaguje hore z jedného z týchto stavov, algoritmus Alpha-beta nemôže orezať žiadnu jej transpozíciu, naviac algoritmus Negascout opísaný v sekcii 2.5 bude na nich vykonať opakované vyhľadávanie pôvodným intervalom. Z týchto dôvodov by bolo dobré zapamätať výsledok prehľadávaní stavu, aby ich transpozície by sme nemuseli znova prehľadávať, ale rovno vrátiť predchádzajúci výsledok. Táto idea je realizovaná s transpozičnou tabuľkou, do ktorej uložíme výsledky prehľadávania stavu. Je indexovaná časťou kľúču generovanej zo stavu hry.
Potrebujeme hashovaciu funkciu, ktorá stavy v strome hry danej hĺbky s veľkou pravdepodobnosťou zahashuje na rôzne hodnoty. Jednu takú funkciu

uviedol Zobrist v článku [12]. Zadefinuje množinu charakteristík, ako napríklad figúrka na danom mieste, hráč na ťahu, možnosti rošád a ťahov En passant. Každý stav hry môže byť presne opísaný podmnožinou týchto charakteristík. Pri vykonaní ťahu niektoré charakteristiky prestanú platiť, iné zase začnú. Ku každej takej charakteristike vygeneruje náhodné číslo. Hashovacia hodnota stavu bude XOR náhodných čísiel tých charakteristík, ktoré platia pre daný stav. Vďaka autoinverznej vlastnosti operátoru XOR, hodnota stavu po vykonaní ťahu môže byť rýchlo vypočítaná z pôvodnej hodnoty a zo zmenených charakteristík.
Hodnoty namerané z predchádzajúceho algoritmu, doplneného transpozičnou tabuľkou sú v tabuľke 2.3.

hĺbka	# listov
1	20,6
2	120
3	1753
4	12098
5	118071

Tabuľka 2.3: Transpozičná tabuľka

2.4 Usporiadanie ťahov

Poradie preskúmaných ťahov je dôležité z pohľadu rýchlosti algoritmu. Ak niektorý ťah má za následok orezanie, je dobré ho prehľadať medzi prvými, tak usporíme prehľadávanie ostatných. Ak ťah nezapríčiňuje orezanie, ale vedie do vysoko ohodnoteného stavu, je tiež dobré ho prehľadať skoro, tak Alpha-beta môže o viac zmenšiť interval, algoritmus Negascout zase nemusí prehľadať podstrom opakovane.
K zoradeniu ťahov sa používajú statické a dynamické metódy. Statické sú založené na teoretických znalostiach šachu. Taká je napríklad idea prehľadávať brania pred nebraním. Samotné brania môžeme ďalej usporiadať heuristikou „najväčšia obeť - najmenší útočník“, podľa ktorého najprv prehľadávame brania najsilnejších oponentových figúrok, a v prípade, že danú figúrku môžeme brať s viacerími z naších, najprv skúsime s tým najslabším.
Dynamické metódy využívajú informácie získané z prehľadanej časti stromu. Také metódy sú napr.:

- **Iteratívne hĺbenie** - prehľadávanie zavoláme najprv do hĺbky jedna, potom na stále väčšie hĺbky tak, že vždy sa prehľadávajú najprv najlepšie ťahy z predchádzajúceho prehľadávania. Tieto ťahy môžeme ukladať do transpozičnej tabuľky. Využívame predpoklad, že k najlepšej hodnote vedie cesta tiež cez dobré hodnoty.
- **Killer-heuristika** - ťah, ktorý zapríčinil orezávanie v niektorom susednom uzle na rovnakej hĺbke stromu hry, prehľadáme skoro. Využíva predpoklad, že dobrý ťah by vo väčšine prípadov zostal rovnako dobrý, aj keby oponent na predchádzajúcej vrstve vykonal iný ťah.
- **History-heuristika** - pre každú štvoricu $[x_1, y_1, x_2, y_2]$ si zapamätáme, že príslušný ťah z $[x_1, y_1]$ do $[x_2, y_2]$ koľkokrát zapríčinil orezanie. Ťahy s vysokou hodnotou prehľadávame skôr.

Namerané hodnoty z algoritmu Alpha-beta, postupne doplneného týmito metódami, používajúce transpozičnú tabuľku sú uvedené v tabuľke 2.4. Vidíme z nich, že aplikovanie iteratívneho hĺbenia môže zväčšiť počet listov. Počet pridaných listov z predchádzajúcich prehľadávaní vo veľkej hĺbke je ale menej, než koľko ušetríme touto metódou.

hĺbka	transpoz. t.	+statické	+iteratívne	+killer h.	+history h.
1	20,6	20,6	20,6	20,6	20,6
2	120	68	90	90	90
3	1753	1025	1065	924	888
4	12098	5160	4636	3174	2913
5	118071	40027	30561	22866	20045

Tabuľka 2.4: Alpha-beta s usporiadanými ťahmi

2.5 Negascout

Alpha-beta vďaka orezávaniam zjavne znížila počet volaní ohodnocovacej funkcie. Ďalšie zrýchlenie môžeme dosiahnuť pomocou prehľadávania s užším intervalom (tzv. *null-window search*). Idea je postavená na predpokladanom dobrom zoradení ťahov. Ak prvý možný ťah je ten najlepší, podstromy ostatných ťahov nemusíme precízne ohodnotiť, stačí ak z nich dostaneme hodnotu, ktorá nás presvedčí, že nemôže byť lepší ako ten prvý a ostatné uzly

čo najskôr orezať. Preto podstrom tohoto ťahu prehľadávame intervalom $(\alpha, \alpha + 1)$. Takéto prehľadávanie hneď odreže podstromy, ak je to možné a vráti hodnotu orezaného podstromu. Ak táto je menšia alebo rovná α, predchádzajúci ťah bol naozaj aspoň tak dobrý a s úzkym intervalom sme ušetrili veľa. Ak návratová hodnota je väčšia ako β, podstrom sa iste odreže a tiež sme ušetrili. Ak hodnota padne do intervalu $[\alpha + 1, \beta)$, potrebujeme presnú hodnotu podstromu, preto ju musíme prehľadať znovu s pôvodným intervalom. Takéto prehľadávania sú časovo náročné, ale predpoklad nám zaisťuje, že tento prípad zriedka nastane, preto úspory na užšom intervale budú väčšie ako náklady na opätovné prehľadávania s celým intervalom.
Algoritmus, využívajúci túto ideu navrhol Reinefield v článku [9] a nazval ho *Negascout*. Je podobný Pearlovmu algoritmu Scout [8], ale je založený na Negamaxovu architektúru. Prehľadávanie opakuje len v prípade, keď výsledok prvého vyhľadávania padne do pôvodného intervalu a je väčší než α, keď nejde o prehľadávanie prvého podstromu (ten sa prehľadáva vždy pôvodným intervalom) a podstromy majú výšku aspoň 3 (v menších podstromoch úzke prehľadávanie vráti rovnaký výsledok ako pôvodný).
Formálny popis algoritmu:

NegaScout(*stav*, *hĺbka*, α, β):integer
if (*hĺbka* = 0 **or** *stav* je koncový) **then**
  return eval(*stav*)
end if
$a = -\infty$;
$b = \beta$;
for all potomok *stavu* **do**
  t = -**NegaScout** (*potomok*, *hĺbka*-1, -b, -MAX(α, a));
  if (($t > \alpha$) **and** ($t < \beta$) **and** ($b \neq \beta$) **and** (*hĺbka* > 1)) **then**
    t = -**NegaScout** (*potomok*, *hĺbka*-1, -β, -t);
  end if
  a = max(a, t);
  if ($a >= \beta$) **then**
    return a;
  end if
  $b = \alpha + 1$;
end for
return a;

Namerané hodnoty z programu `negascout.exe` používajúce metódy opísané v predchádzajúcej sekcii sú v tabuľke 2.5.

hĺbka	# listov
1	20,6
2	90
3	662
4	2279
5	14211

Tabuľka 2.5: Negascout

2.6 Horison effect

Doteraz opísané algoritmy prehľadávajú strom hry do danej hĺbky, tam sa zastavia, ohodnotia stav a získanú hodnotu propagujú hore. Nevýhodou týchto algoritmov je, že na poslednej vrstve môžu vybrať ťah, ktorý má nepríjemné dôsledky. Uvažujme o situácii, keď na predposlednej vrstve sú dva možné ťahy: posun na prázdne políčko, a branie súperovej figúrky dámou. Algoritmy druhú možnosť ohodnotia lepšie a jej hodnotu propagujú nahor. Ale ak súper chránil branú figúrku, jeho odpoveď bude branie našej dámy. Reálna hodnota nášho vybraného ťahu je teda horšia a algoritmus by mal vybrať tú druhú možnosť. Predísť takýmto chybám by sme mohli zvýšením hĺbky prehľadávania, tá by ale exponenciálne zväčšila zložitosť, a chyba by sa naďalej mohla vyskytnúť o vrstvu nižšie.
Dôvodom tohoto javu je, že ohodnotené stavy hry nemusia byť stabilné: ďalší vykonaný ťah môže oveľa zmeniť ich hodnotu. Potrebovali by sme rozšíriť prehľadávanie tak, aby ohodnocovacia funkcia bola zavolaná len na stabilné stavy. K tomu nám vyhovuje Shannonov algoritmus typu B, ktorý prehľadáva len po silných ťahoch. Keď žiadny silný ťah nenájde, stav je stabilný, zavolá ohodnocovaciu funkciu.
Za silné ťahy sa obvykle považujú brania figúrok, promócie a odskoky z šachu. Takých ťahov je pomerne málo a reťazce z nich sú krátke, preto prehľadávanie ich podstromov zhorší celkový algoritmus o menej, ako pridanie jednej vrstvy. Tabuľka 2.6 obsahuje počet listov hracieho stromu prehľadávaného algoritmom Negascout bez a s rozšíreným prehľadávaním.

hĺbka	bez rozš. prehľ.	s rozš. prehľ.
1	20,6	22,2
2	90	112
3	662	734
4	2279	2528
5	14211	14841

Tabuľka 2.6: Rozšírené prehľadávanie

2.7 Ohodnotenie stavov

Doteraz opísané algoritmy umožnili, aby sme čo najviac stavov preskúmali čo najrýchlejšie. Ozajstnú silu šachového enginu dáva správne ohodnotenie stavov v listoch prehľadaného stromu hry. Také ohodnotenie je založené na hlbokých teoretických znalostiach šachu. V tejto sekcii ukážeme niekoľko faktorov, ktoré môžeme brať do úvahy pri zostrojení ohodnocovacej funkcie. Základom ohodnotenia stavu je súčet hodnôt figúrok na šachovnici. Vzájomná hodnota figúrok je dôležitá, aby sme mohli správne rozhodnúť pri výmenách. Najpoužívanejší pomer hodnôt figúrok pešiak 1, jazdec 3, strelec 3, veža 5, dáma 9, rovnaký, ako definoval Shnannon v [10]. Kráľ sa obyčajne ohodnotí rádovo vyššou hodnotou, aby jeho strate program skúsil zabrániť. Existujú iné systémy pomerov hodnôt, ktoré ohodnotia strelca a jazdca vyššie, aby predišli ich výmene za troch pešiakov.
Pomer hodnôt sa môže zmeniť počas hry dynamicky:

- Klesaním počtu pešiakov jazdec stratí zo svojej hodnoty, lebo jeho sila je skrytá v tom, že môže preskakovať súperove blokujúce figúrky.
- Naopak veža tým získa hodnotu, lebo ona je efektívna, ak má na tabuľke veľa voľného priestoru.
- Jeden strelec môže ovládnuť len políčka jednej farby a tak sa dá ľahko spred neho uniknúť. Preto keď druhého strelca odoberú, zostávajúci stratí zo svojej sily a tak i z hodnoty.
- Figúrky s vysokou mobilitou (počtom možných ťahov) sú silnejšie než s nízkou, lebo tým majú kontrolu nad cieľovými políčkami. Do mobility je možné pripočítať aj políčka, kam by daná figúrka mohla skočiť, ale stojí tam figúrka rovnakej farby - tým je chránená.

Ohodnotenie figúrok ďalej môže závisieť na ich pozíciách na tabuľke:

- Pešiaci na stĺpcoch *d* a *e* by mali čo najskôr postupovať dopredu, aby ovládali stred tabuľky a zároveň neblokovali strelcov. Preto na pôvodných pozíciách dostanú nízku hodnotu, po skoku dopredu vyššiu. Naopak, pešiakom na krajných stĺpcoch sa oplatí zostať na pôvodných pozíciach, aby chránili kráľa po rošáde. Oni tam dostanú vyššiu hodnotu ako po skoku. Keď už pešiak bol posunutý, za každý ďalší posun môže dostať stále vyššiu hodnotu, aby s tým bol stimulovaný k promócii.

- Strelec a jazdec by sa mali vyhýbať okrajom tabuľky, kde ich možný počet ťahov je nízky a tým stratia zo svojich síl. Takisto by mali zo svojich pôvodných pozícií čo najskôr odskočiť, aby uvoľnili kráľovi priestor pre rošádu. Tak ich hodnoty budú vysoké v strede tabuľky, nízke na okrajoch a najnižšie na pôvodných pozíciách.

- Kráľ má zostať v bezpečných pozíciách, ideálne po rošáde v skrytí pešiakov.

Tieto zmeny hodnôt sa dajú naimplementovať tabuľkami lokálnych hodnôt (tzv. piece-square tables), ktoré pre danú figúrku a danú pozíciu obsahujú zmenu jej ohodnotenia.
V tabuľke 2.7 sú uvedené výsledky zápasov medzi algoritmami s rôznymi ohodnocovacími funkciami. Samotné ohodnotenie materiálu (figúrok) oproti ostatným ohodnoteniam je najslabšie. Najlepšie fungujúca metóda je použitie tabuľky lokálnych hodnôt.

hráč 2 \ hráč 1	A	B	C	D	E	F	G	H
A (materiál)	-	1	1	1	1	1	1	1
B (mat. + piece-square)	2	-	2	X	2	1	2	1
C (mat. + mobilita)	2	1	-	2	2	1	1	2
D (mat. + mob. + P.S.)	2	1	2	-	2	X	2	2
E (mat. + dynamic.)	2	1	1	1	-	1	2	1
F (mat. + dyn. + P.S.)	2	1	1	1	2	-	1	X
G (mat. + dyn. + mob.)	2	2	X	2	2	2	-	1
H (mat. + dyn. + mob. + P.S.)	2	1	X	X	2	X	2	-

Tabuľka 2.7: Úspešnosť ohodnocovacích funkcií proti sebe

Kapitola 3

Podobné projekty

V tejto kapitole sa zoznámime s programami, ktoré môžu byť použité pri vývoji šachového enginu.

3.1 XBoard

Xboard [11] (pod Windowsom WinBoard) je multiplatformové grafické rozhranie pre šach. Môže byť pripojený k rôznym šachovým engineom, ktoré komunikujú cez Chess Engine Communication Protocol. Môže byť pripojený k internetovým hracím serverom a podporuje e-mailové korešpondenčné zápasy. Nepodporuje ale hru medzi viacerými umelými hráčmi, a komunikačný protokol je tiež obmedzený na malý počet šachových variantov.

3.2 Arena

Arena [1] je grafické rozhranie pre šach, zamierené na porovnanie šachových enginov. Pre komunikáciu s enginmi podporuje Chess Engine Communication Protocol a Universal Chess Interface. Chýba ale podpora šachových variantov.

3.3 General Game Playing

Iný smer vývoja sa nezameriava na dokonalý algoritmus pre jednu danú hru, ale na algoritmus schopný hrať čo najviac hier. Hry sú zadané množinou pra-

vidiel a General game player [4] musí rozpoznať, ako hru správne a efektívne hrať.

3.4 ChessNet

ChessNet je viachráčová, sieťová implementácia doskových hier s klient-serverovou architektúrou, ktorú som v rámci ročníkového projektu vytvoril s cieľom, aby tvorila základ bakalárskej práce. Podporuje rôzne typy hier, ktoré k nemu môžu byť pridané vo forme JavaScriptových súborov, obsahujúce pravidlá hier. Podporuje taktiež umelú inteligenciu, ktorá môže byť pridaná vo forme externých aplikácií. Aplikácie môžu byť parametrizované, aby splnili potreby používateľov na rôznej úrovni. Podporuje zápasy medzi umelými hráčmi. Je vhodný pre testovanie rôznych stratégií v hrách.
K programu bolo potrebné pridať sofistikovanejší spôsob testovania umelých hráčov.

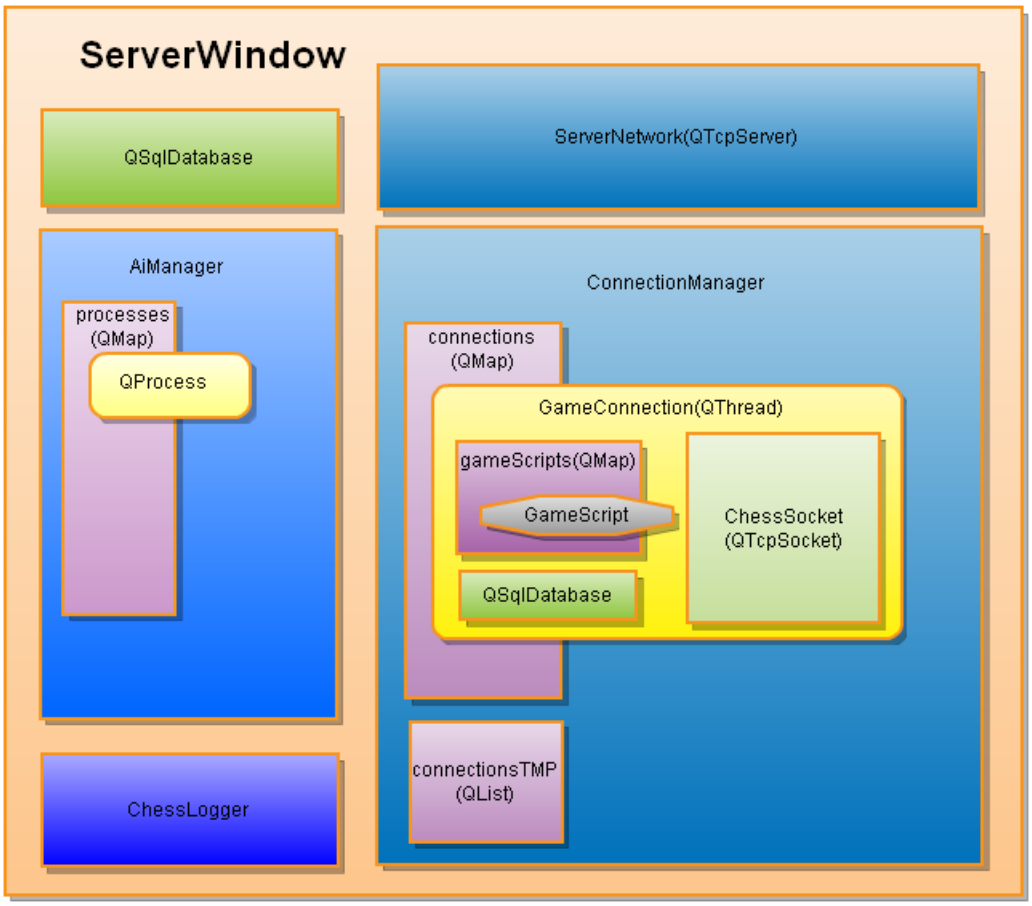
Kapitola 4

Programátorská dokumentácia

ChessNet je sieťová implementácia tabuľových hier typu šachy s umelou inteligenciou. Je vyvíjaný v prostredí Visual Studio 2008 pod Windows XP. Je postavený na multiplatformový toolkit Qt. Má klient-serverovú architektúru. Klient jednoducho zobrazuje aktuálny stav hry a pošle serveru správy od užívateľa. Server sa postará o pravidlá hier, o komunikáciu medzi užívateľmi, o uloženie zmien do databázy, a o volanie umelej inteligencie. Hry môžu byť pridané vo forme skrípt, umelá inteligencia je počítaná externými aplikáciami.
K prekladaniu zo zdrojového kódu je potrebné mať:

- knižnice Qt (používaná verzia 4.5.1)
- knižnice MySQL (používaná verzia 5.1.46)

Štruktúra serveru je ilustrovaná na obrázku 4.1.Pozostáva zo štyroch hlavných častí: `ServerNetwork`, `ConnectionManager`, `AiManager` a `ChessLogger`.
`ServerNetwork` je potomok triedy QTcpServer, ktorý poslúcha na danom porte. Pri pripojení hráča si vytvorí socket, cez ktorý sa potom vedie komunikácia s hráčom. Údaje o tomto socketu obsahuje QTcpServer, a sú identifikovateľné tzv. SocketDescriptorom. Pripojenie hráča `ServerNetwork` hlási časti `ConnectionManager` poslaním tohto identifikátora.
`ConnectionManager` je trieda spravujúca pripojených užívateľov. Po obdržaní správy od ServerNetworku, vytvorí inštanciu triedy `GameConnection`, ktorý je potomkom QThread. Je to vlákno, v ktorom budú behať všetky výpočty vzťahujúce sa k danému užívateľovi a bude obsahovať aj socket, ktorý si vytvorí zo SocketDescriptoru. `ConnectionManager` si zaradí túto inštanciu do kontajneru tmpConnections, ktorý obsahuje pripojených, ale



Obr. 4.1: Štruktúra serveru

zatiaľ neprihlásených hráčov. Ak hráč pošle svoje prihlasovacie údaje, ktoré sú správne, pripojenie bude premiestnené do kontajneru connections, ktorý je asociatívne pole, mapované podľa userID - identifikátor užívateľa. Každý užívateľ môže byť naraz prihlásený iba s jedným klientom.
V prípade viacerých pripojených hráčov `ConnectionManager` podáva správy medzi pripojeniami, alebo medzi pripojením a `AiManagerom`. Ak sa hráč odpojí, `ConnectionManager` vymaže jeho pripojenie.
`GameConnection` je trieda reprezentujúca pripojeného užívateľa. Je potomkom triedy QThread, obsahuje informácie o užívateľovi, aktuálne stavy jeho otvorených hier a sieťové pripojenie `ChessSocket`.
`ChessSocket` čaká správy od klienta a spracuje ich. Pri požiadavke o novú hru načíta skript zo súboru, informácie o novej hre uloží a pošle klientovi.

Pri požiadavke o otvorenie staršej hry na načítanom skripte uplatňuje aj doterajšie ťahy, a informácie o aktuálnom stave pošle klientovi. Pri novom ťahu uplatňuje zmeny v skripte, stav uloží do databázy a pošle súperovi.
`AiManager` je trieda starajúca sa o umelú inteligenciu. Pri požiadavke zavolá externú aplikáciu, pošle jej aktuálny stav hry a čaká, kým aplikácia vráti nový stav - stav po vybranom ťahu. Ten stav potom uloží a pošle súperovi.
`ChessLogger` podáva ostatným častiam programu službu logovania. Pri spustení programu vytvorí logovací súbor, do ktorého potom zapíše všetky logovacie správy, vrátane času i odosielateľa. Slúži k ľahšiemu debugovaniu.

Uloženie dát

Na uloženie dát používa server pripojenie k databáze MySQL. V databáze má uložené informácie o typoch hier, o botoch, o užívateľoch a o zápasoch. Štruktúra databázy:

Tabuľka `users` obsahuje informácie o užívateľoch:

UserID	ID užívateľa
UserName	prihlasovacie meno užívateľa
Password	heslo užívateľa
Description	popis

Tabuľka `gametypes` obsahuje informácie o typoch hier:

GameTypeID	ID typu hry
GameTypeName	názov typu hry
ScriptFile	súbor obsahujúci skript hry
GameTypeDescription	popis hry

Tabuľka `bots` obsahuje informácie o botoch:

BotID	ID bota
BotName	meno bota
BotDescription	popis
FileName	súbor obsahujúci externú aplikáciu
Parameters	parametre k volaniu externej aplikácie
GameTypeID	ID typu hry, ktorej je bot určený

Tabuľka `games` obsahuje informácie o zápasoch:

GameID	ID zápasu
GameTitle	názov zápasu
GameTypeID	ID typu hry
Player1ID	ID bieleho hráča
Player2ID	ID čierneho hráča
LastSender	ID odosielateľa poslednej správy
LastTime	čas odosielania poslednej správy
Status	popis stavu hry
BotIdle	`1` ak bot tejto hry práve nebeží,inak `0`

Tabuľka `gameevents` obsahuje hracie udalosti:

EventID	ID udalosti
EventTime	čas udalosti
GameID	ID hry udalosti
EventType	typ udalosti(`data` alebo `text`)
Sender	ID odosielateľa
EventData	obsah udalosti

Podporované hry

Program podporuje šachovnicové hry pre dvoch hráčov. Šachovnica je dvojrozmerné pole ľubovoľnej veľkosti. Hráči nemusia striedavo ťahať, ale vždy musí byť presne určené, ktorý hráč je na ťahu. Pravidlá hier sú uložené v skriptách typu `ECMAScript` - je to štandardizovaná verzia JavaScriptu. Skripta sa načíta zo súboru, triedou QScriptEngine sa interpretuje a vrátená QScriptValue hodnota sa uloží do `GameConnection`-u hráča. Táto hodnota bude obsahovať vždy aktuálny stav hry a bude vyhodnocovať správnosť nových ťahov.
Musí obsahovať nasledujúce položky:

- `gamestate` - objekt obsahujúci aktuálny stav hry. Jeho obsah sa uloží do databázy. Jediná jeho povinná položka je
  - `onmove` - Integer, číslo hráča na ťahu(1 alebo 2), alebo 0 na konci hry.
- `setGameState(gs)` - funkcia, ktorá nastaví hodnotu objektu `gamestate` na `gs`. Parameter musí byť typu Objekt.
- `startNewGame` - funkcia, ktorá naplní gamestate s informáciami novej hry
- `clientData` - funkcia, vráti objekt obsahujúci aktuálny stav hry vo forme vhodnej pre klienta. Tento objekt má položky:
  - `rows` - Integer, počet riadkov na tabuľke
  - `columns` - Integer, počet stĺpcov
  - `boarddata` - String dĺžky rows*columns, obsahuje tabuľu ako vidí daný hráč
  - `lastmove` - String vo forme „$x1 - y1[-x2 - y2[...]]$“, obsahujúci súradnice posledného ťahu
  - `lastmovetext` - String obsahujúci čitateľné informácie o poslednom ťahu
  - `orientation` - String, indikátor pre klienta o orientácii tabuľky. Ak má hodnotu „`switched`“, klient nakreslí tabuľku otočene.
  - `showcaptions` - String, indikátor pre klienta o zobrazení číslovania tabuľky. Ak má hodnotu „`true`“, číslovanie sa zobrazí
- `isLegal(move)` - funkcia, vráti Boolean TRUE, ak ťah move je v aktuálnom stave správny, inak vráti FALSE. Parameter move je String tvaru „$x1 - y1[-x2 - y2[...]]$“.
- `legalMoves` - funkcia, vráti String, zoznam všetkých správnych ťahov daného hráča v danom stave. Ťahy majú formu „$x1-y1[-x2-y2[...]]$“ a sú oddelené čiarkou.
- `makeMove(move)` - funkcia, nastaví gamestate na stav, ktorý nastane po uplatnení ťahu move. Parameter move je String tvaru „$x1-y1[-x2-y2[...]]$“.

- `status` - funkcia, vráti String, krátku vetu o aktuálnom stave hry.
- `showDebug` - Boolean, indikátor pre server, či má logovať zo skriptu
- `debug` - String, obsahujúci logy

Umelá inteligencia

Umelú inteligenciu zabezpečuje trieda `AiManager`. Ak dostane správu, že na ťahu je umelý hráč, z databázy zistí, ktorú externú aplikáciu treba zavolať a s akými parametrami. Vytvorí z nej inštanciu triedy QProcess. Umelú inteligenciu počítajú aplikácie, ktoré na vstup dostanú ID a aktuálny stav hry, a na výstup pošlú príkazy v tvare:

- MOVE $x1 - y1[-x2 - y2[\ldots]]$ - programom vybraný ťah
- TEXT message - textová správa určená pre hráča

Sieťová komunikácia

Sieťová komunikácia prebieha posielaním objektov typu `ChessMsg`, ktorý je asociatívne pole s kľúčmi aj hodnotami typu QString. Tento objekt má povinný kľúč type, ktorý určí jeho typ. Ostatné kľúče závisia od typu. Priebeh komunikácie je zobrazený na obrázku 4.2.

Obsah ChessMsg podľa typu:

type `login`: pošle server po pripojení, obsahuje prihlasovacie údaje

type	`"login"`
login	prihlasovacie meno užívateľa
password	heslo

type `loadgame`: pošle klient, ak užívateľ chce pokračovať hru

type	`"loadgame"`
gameID	ID vybranej hry

Obr. 4.2: Priebeh sieťovej komunikácie

type `gameData`: pošle server, obsahuje udalosti danej hry

type	`"gameData"`
gameID	ID hry
player1ID	ID prvého hráča
player2ID	ID druhého hráča
gameTitle	názov hry
eventType*1..n*	typ udalosti(„text“ alebo „data“)
eventData*1..n*	obsah udalosti(textová správa alebo stav hry)
eventTime*1..n*	čas udalosti
sender1..n	meno odosielateľa udalosti
eventCount	počet udalostí
legalMoves	zoznam správnych ťahov hráča v danom stave hry
statusText	popis stavu hry

type `newgame`: pošle klient, ak užívateľ chce začať novú hru

type	`"newgame"`
gameTypeID	ID typu novej hry
player1ID	ID bieleho hráča
player2ID	ID čierneho hráča
myColor	farba odosielateľa(1-biela,2-čierna)
gameTitle	vybraný názov hry
gameCount	počet hier v AI režimu

type `playerData`: pošle server po prihlásení, obsahuje informácie o hrách užívateľa

type	`"playerData"`	
myUserID	ID užívateľa	
myUserName	meno užívateľa	
gameID*1..n* gameTitle*1..n* gameTypeName*1..n* player1Name*1..n* player2Name*1..n* lastName*1..n* status*1..n* startedGamesCount	ID zápasu názov zápasu typ hry meno bieleho hráča meno čierneho hráča meno odosielateľa poslednej správy popis stavu hry počet začatých hier	údaje o začatých hrách
newGameTypeID*1..n* newGameTypeName*1..n* newGamesCount	ID typu hry meno typu hry počet typov hier	údaje o možných typoch novej hry
botID*1..n* botName*1..n* botCount	ID botu meno botu počet botov	údaje o botoch
opponentID*1..n* opponentName*1..n* opponentCount	ID užívateľa meno užívateľa počet užívateľov	údaje o protihráčoch

type `textmessage`: pošle klient, ak užívateľ chce poslať textovú správu

type	`"textmessage"`
text	text správy
gameID	ID hry

type `movemessage`: pošle klient, ak užívateľ urobil ťah

type	`"movemessage"`
move	Vybraný ťah vo forme $x1 - y1[-x2 - y2[\ldots]]$
gameID	ID hry

type `textNotification`: pošle server interne, hlási tým textovú správu súperovi

type	`"textNotification"`
gameID	ID hry
eventType0	`"text"`
eventData0	obsah udalosti(textová správa)
eventTime0	čas udalosti
sender0	meno odosielateľa udalosti
eventCount	počet udalostí("1")

type `moveNotification`: pošle server interne, hlási tým vykonaný ťah súperovi (ak súper je bot, dostane správu len v prípade, že je na ťahu)

type	`"moveNotification"`
gameID	ID hry
eventType*1..n*	typ udalosti(`"data"`)
eventData*1..n*	obsah udalosti(stav hry)
eventTime*1..n*	čas udalosti
sender*1..n*	meno odosielateľa udalosti
eventCount	počet udalostí
onMove	ID hráča na ťahu

Kapitola 5

Užívateľská dokumentácia

Inštalácia servera

Server programu ChessNet k správnemu fungovaniu potrebuje pripojenie k databázovému serveru MySQL, k webserveru a otvorený port na firewalle. K efektívnemu behu je odporúčané mať databázový server na rovnakom počítači ako ChessNet server. Odporúčaná, jednoduchá, voľná distribúcia obsahujúca všetky potrebné veci je *Wampserver*. Po nainštalovaní treba v ňom vytvoriť databázu, s ktorou ChessNet bude pracovať. K tomu v ľubovoľnom prehliadači na adrese `http://localhost/phpmyadmin/` do textového poľa zadáme vybraný názov databáze (napr. `chessdb`). Po vytvorení databázy je odporúčané vytvoriť užívateľa databázy, ktorý má právo pracovať s novovytvorenou databázou, ale s ničím iným. K tomu v hornom riadku klikneme na položku `Privileges` a potom na „`Add a new user`“. Na novej stránke zadáme vybrané užívateľské meno a heslo a vyberieme predtým vytvorenú databázu pre tohoto užívateľa. Tieto nové údaje (názov DB, meno a heslo) budeme potom používať pri spustení servera.
Ak chceme povoliť registráciu nových užívateľov, musíme nakopírovať adresár `www` od šachového serveru do inštalačného adresára Wampserveru a v `db.php` zadať prihlasovacie údaje k databáze. Takto keď pri spustení servera zadáme verejnú adresu počítača ako adresu webového servera, užívatelia budú schopní sa zaregistrovať do hry cez webové rozhranie.

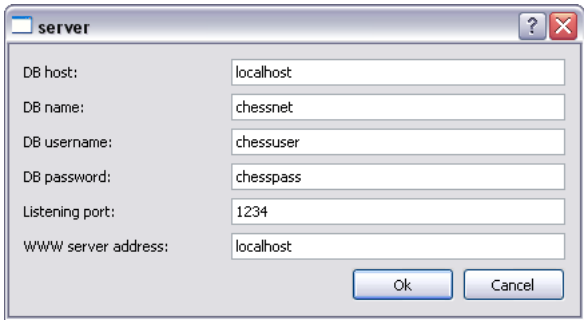
Administrácia servera

Po spustení súboru `server.exe` sa aplikácia dostane do offline stavu. V takom stave sa k nej nemôžu pripojiť klienti. K prechodu do stavu online musíme kliknúť na tlačítko `Listen`. V novom dialógovom okne (obrázok 5.1) nastavíme informácie o databáze (adresa servera, názov DB, meno a heslo) a o sieti (port, cez ktorý sa klienti pripojujú a adresa webového servera). Ak sa aplikácia s danými nastaveniami úspešne pripojí k databáze a otvorí poslúchací port, dostaneme sa do online stavu a nastavenia budú uložené, aby nabudúce už nemuseli byť znova vyplnené. V takom prípade sa aplikácia už hneď po spustení pripojí k databáze a pre prechod do online stavu stačí kliknúť na tlačítko Listen.
V stave online sa k serveru môžu pripojiť klienti. Na obrázku 5.3 vidíme okno servera v takom stave. Každého pripojeného hráča uvidíme v pravej časti okna s názvom `Connections`.
Úpravy v databáze môžu byť vykonané v ľavej časti okna. Ak je server pripojený k databáze, uvidíme tu dve tabuľky: tabuľku typov hier a tabuľku s botmi. S tlačítkami nad tabuľkami môžeme pridať, editovať a odobrať jednotlivé skripty typov hier (obrázok 5.2) a jednotlivé boty (obrázok 5.4). Skripty sú uložené v adresári `scripts`. Boty sú v adresári `bots` spolu s textovými súbormi, opisujúcimi ich parametry. Pri pridaní botu máme vybrať ku ktorej hre patrí, preto je treba najprv pridať jeho hru.
Ak si chceme pozrieť jednotlivé zápasy, musíme sa prihlásiť s klientom ako užívateľ *admin* s heslom *toor*. Takto dostaneme prehľad všetkých zápasov na serveri, a tiež možnosť začať zápasy typu bot-bot. Pre zrýchlenie testovania botov je možné vytvoriť naraz viac zápasov medzi nimi. Po načítaní vytvorených zápasov s tlačítkami „`Force bot`“ a „`Pause bot`“ môžeme spustiť a zastaviť bota na zistenie nasledujúceho ťahu.
Zoznam aktuálne pracujúcich botov je v pravom časti serverového okna s názvom `Active bots`.

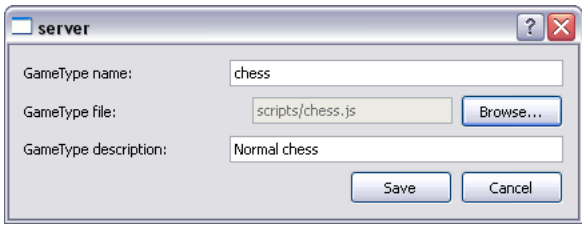
Používanie klienta

Po spustení programu `ChessClient.exe` vyplníme svoje prihlasovacie údaje a informácie o serveri. V prípade, že ešte nemáme registráciu, alebo chybne sme vyplnili prihlasovací formulár, objaví sa spáva chybného prihlásenia s možnosťou registrácie (obrázok 5.5). Ak administrátor servera to povolil (viz. inštalácia servera), dostaneme sa na webovú stránku, kde sa môžeme zare-

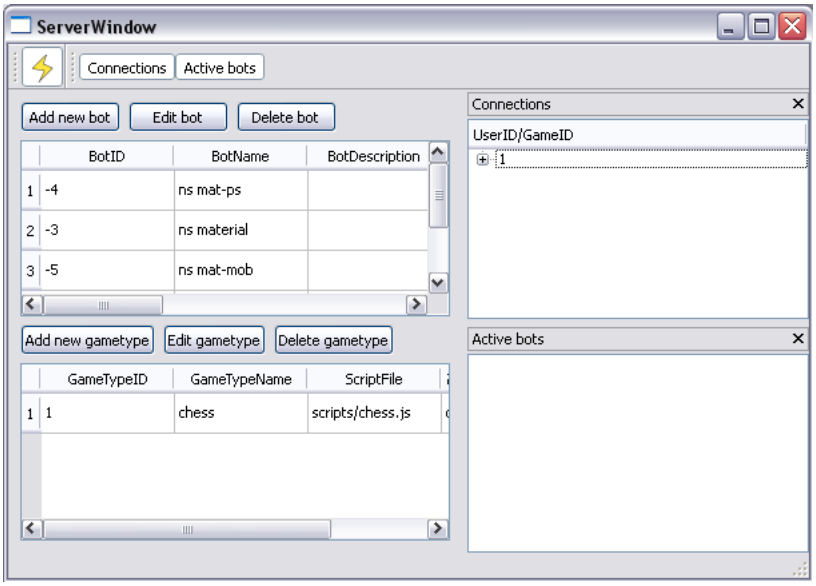
gistrovať.
Ak sa prihlasovanie úspešne uskutoční, zobrazí sa nám zoznam našich zápasov, a zoznam nastavení pre spustenie nového zápasu (obrázok 5.6). Ak si vyberieme pokračovanie alebo nový zápas, v novom tabu sa zobrazí hracia plocha, a chatové okienko (obrázok 5.7). Tu si môžeme pozrieť doterajší postup hry, posielať správy súperovi, alebo vykonať nové ťahy. Pre vykonanie ťahu si najprv kliknutím vyberieme figúrku. Ak má figúrka legálne ťahy, tie sa hneď zobrazia a kliknutím na cieľové políčko potvrdíme ťah. V prípade, že sme sa rozhodli ťahať inou figúrkou, predchádzajúci výber môžeme zrušiť kliknutím na políčko, kam vybraná figúrka nemôže skočiť.
Ak chceme ukončiť hru, okno jednoducho zavrieme.



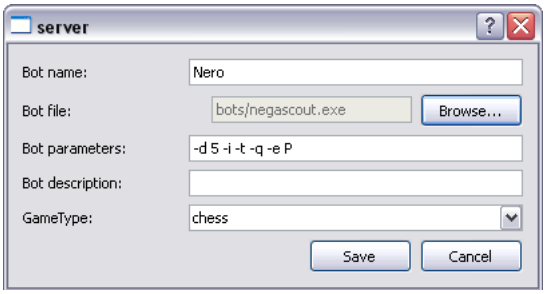
Obr. 5.1: Formulár o nastavení servera



Obr. 5.2: Pridanie skripta typu hry

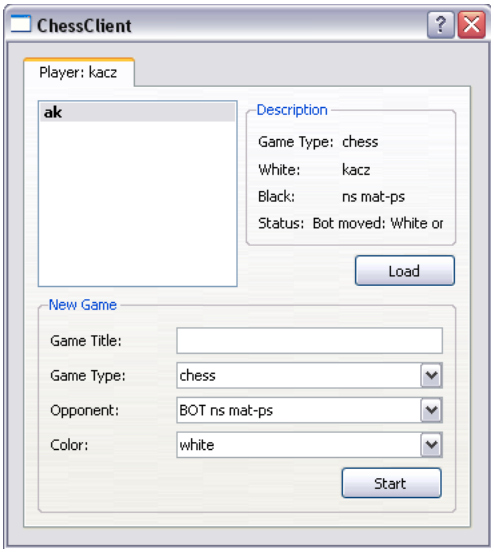


Obr. 5.3: Okno servera

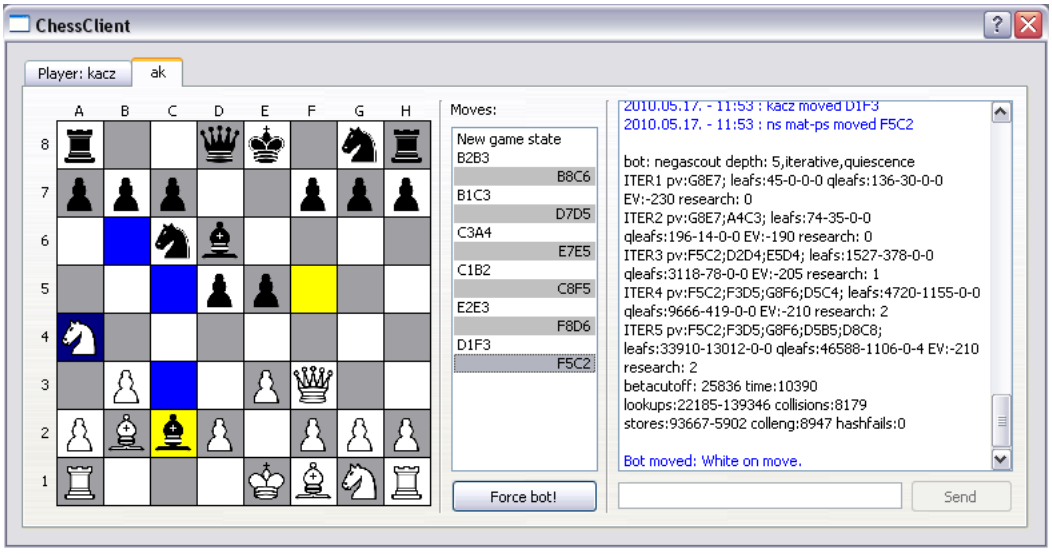


Obr. 5.4: Pridanie botu

Obr. 5.5: Správa chybného prihlásenia



Obr. 5.6: Zoznam hier



Obr. 5.7: Hracia plocha

Kapitola 6

Záver

Cieľom práce bolo vytvorenie šachového enginu pre prostredie ChessNet a v rámci vývoja porovnať šachové stratégie. V druhej kapitole sme ukázali teóriu tejto problematiky a pomocou opísaných algoritmov sme postupne vytvorili časovo efektívnejšie enginy. Ukázali sme spôsoby na zvýšenie kvality hry a porovnali sme ich.
Vytvorený engine je schopný zmysluplne zahrať šach na vyššej úrovni ako sám autor. Nie je ale porovnateľný enginmi, ktoré sa vyvíjajú dlhé roky a sú založené na rozsiahlych štatistických údajoch.
Ako slabý bod programu ChessNet môže byť spomenutá neintuitívna inštalácia (práca s phpMyAdmin, spustenie webového rozhrania). Databázový server MySQL bol vybraný z dôvodu dobrej kompatibility s rozšírenými technológiami. V ňom uložené dáta môžu byť ľahko spracované a prezentované napr. na webe. Ďalšia práca by tak mohla byť zameraná na doplnenie webbového rozhrania napríklad AJAX-ovým klientom.
Program ChessNet bol prispôsobený k porovnaniu enginov. Vďaka podpory rôznych doskových hier, ako napr. dáma, píškvorky a rôzne varianty šachu, ďalšia práca môže byť zameraná aj na vývoj a porovnanie stratégií pre tieto hry.

Literatúra

[1] Arena 2.0.1 Chess GUI - http://www.playwitharena.com/

[2] http://www.chessbase.com/newsdetail.asp?newsid=1574

[3] Garry Kasparov beat Deep Blue (Computer) 4 to 3 - http://www.chessgames.com/perl/chess.pl?pid=29912&pid2=15940

[4] General Game Playing - http://games.stanford.edu

[5] D. E. Knuth and R. W. Moore (1975). *An analysis of alpha-beta pruning* Artificial Intelligence **6**, 293–326.

[6] E.M. Landis, I.M. Yaglom, Remembering A.S. Kronrod, English translation by Viola Brudno. W. Gautschi (ed.) [written for Uspekhi Matematicheskikh Nauk, English publication Math. Intelligencer (2002), 22-30], available at Stanford University School of Engineering SCCM-00-01 (PostScript). Retrieved on 19 December 2006

[7] Von Neumann and Morgenstern (1944), *Theory of Games*, 125, Princeton

[8] Pearl, J. (1980): *Scout: A Simple Game-Searching Algorithm with Proven Optimal Properties*. Proceedings of the First Annual National Conference on Artificial Intelligence. Stanford

[9] Reinefeld, A. (1983). *An Improvement to the Scout Tree-Search Algorithm*. ICCA Journal, Vol. **6**, No. **4**, 4-14.

[10] C.E. Shannon (1950), *Programming a Computer for Playing Chess*, Philosophical Magazine, Ser. **7**, Vol. **41**, Bell Telephone Laboratories, March 1950

[11] Xboard GUI - http://www.gnu.org/software/xboard/

[12] Zobrist, A.L. (1970). *A New Hashing Method with Application for Game Playing*. Technical Report 88, Computer Science Department, The University of Wisconsin, Madison, WI, USA. Reprinted (1990) in ICCA Journal, Vol. 13, No. 2, pp. 69-73. ISSN 0920-234X.

Dodatok A

Obsah CD

Súčasťou práce je priložený CD-ROM. Jeho obsah je nasledovný:

- V adresári `ChessNet` sa nachádzajú dva súbory: `bin.zip` obsahuje binárnu distribúciu programu ChessNet pre Windows spolu s botmi a so skriptami hier. Súbor `src.zip` obsahuje zdrojové súbory projektu.
- Adresár `soft` obsahuje použité knižnice (Qt 4.5.1, MySQL 5.1.46) a distribúciu databázového servera WampServer.
- Súbor `thesis.pdf` obsahuje text tejto práce.