Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Bakalářská práce

Implementace herních strategií

Plzeň 2012 Petr Pavlíček

Prohlášení

Prohlašuji, že jsem bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 10. května 2012

Petr Pavlíček

Abstract

This work explores possibilities of the game strategies. It seeks for a way how to learn computers to play a specific game. For this purpose, the two players board game Othello was chosen (also known as Reversi). The software system was created in ObjectPascal language, which uses the negamax algorithm with alpha-beta pruning as the game strategy implementation. It implements three types of players: a human player, computer player and network player. The network player uses TCP protocol for the communication.

Obsah

1 Úvod

Hry byly a jsou součástí lidského života již od nepaměti. Hraní her přináší člověku odpočinek, zábavu, ale i možnosti sebezdokonalování. Hra je taktéž společenská záležitost, ke které často potřebujeme spoluhráče nebo protihráče. V dnešním světě se jím může stát i počítač.

Počítač ale neumí hrát hry sám o sobě. Stejně jako člověk se musí naučit pravidla hry. Ty samotné ale k výhře často nestačí. Ještě je třeba zapojit strategii, kterou si člověk osvojuje dlouhodobým hraním her. Také pro počítač existují herní strategie, z nichž některé jsou obecně použitelné.

Jedním z cílů této práce je prozkoumat tyto herní strategie. Zjednodušeně řečeno, nalézt možnosti, jak naučit počítač hrát takovou hru, u které lze nějakou strategii uplatnit. Dalším cílem je zjištěné poznatky použít pro implementaci konkrétní hry. Tou byla zvolena klasická desková hra pro dva hráče Othello, též známá jako Reversi.

2 Teoretická část

2.1 Hry

Jednoduché hry pro jednoho či více hráčů spadají do kategorie her v rozšířeném tvaru. Ty se vyznačují tím, že existuje určitý počáteční stav a řada konečných stavů. Mezi těmito stavy existují další mezistavy, což jsou jednotlivé tahy hráčů. Tyto hry můžeme dále dělit na:

1. hry s úplnou informovaností (klasické deskové hry - dáma, šachy...)
2. hry s neúplnou informovaností (např. karetní hry, deskové hry s hodem kostkou...)

U obou kategorií platí, že hráč má informaci o tom, jaké tahy provedl protihráč v minulých tazích. U her s neúplnou informovaností ale například nevíme, jaké karty mají protihráči v ruce nebo jakou kartu získáme v dalším kole z balíku. Je do nich přidán prvek náhody. Naopak u her s úplnou informovaností máme po celou dobu hraní naprostý přehled o stavu hry, není zde žádný prvek náhody. Je tedy možné odhadovat tahy soupeře a promýšlet tahy dopředu.

U her s neúplnou informovaností se nemá moc smysl zabývat nějakou globální strategií, která by byla použitelná na všechny typy těchto her. Pokud lze nějakou strategii nalézt, pak bude použitelná právě pouze pro danou hru. U některých her ani žádná strategie neexistuje. Při implementaci takovýchto her záleží hlavně na správné interpretaci pravidel hry. Naopak strategie se uplatní u her s úplnou informovaností, které budou obsahem další kapitoly.

2.2 Strategie u her s úplnou informovaností

Jednotlivé tahy her s úplnou informovaností jsou stavy, kdy se z výchozího stavu dostaneme do konečného stavu. Tyto stavy se mění na základě přípustných tahů, které se řídí podle pravidel hry. U každé takovéto hry bychom mohli vytvořit jakýsi graf, který by reprezentoval všechny možné stavy hry od zahájení až po konec hry. Takovému grafu se říká úplný strom hry. Jeho kořenem je výchozí stav hry, uzly jsou jednotlivé stavy hry a listy jsou konečné stavy hry. Hrany představují přípustné tahy. Úplný se nazývá právě

proto, že obsahuje všechny možné stavy hry. V praxi je ale málokdy možné takový strom vytvořit, neboť by byl příliš složitý. Proto se spíše používá neúplný herní strom, který obsahuje pouze omezený počet tahů, odpovídajících hloubce stromu. Listy tohoto stromu již zdaleka nemusí být konečnými stavy. Musíme proto tyto listy ohodnotit nějakou hodnotou, která nám určí pravděpodobnost, že tento stav povede k vítězství (konečnému stavu).

Strategie pak spočívá v prohledávání těchto stromů a vyhledání optimální cesty. Několik algoritmů pro prohledávání těchto stromů budou popsány dále. V těchto kapitolách bylo čerpáno z [Vopravil(2007)] a z [Russell(1995)].

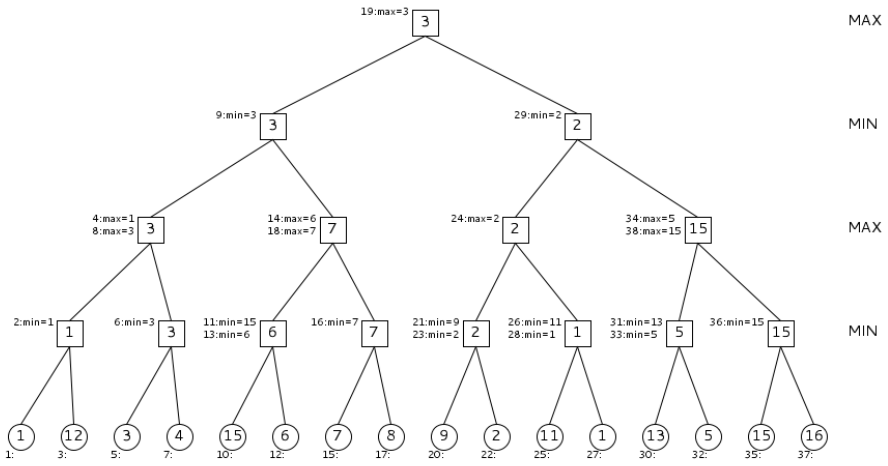
2.2.1 Minimax

Minimax je algoritmus vyhledávání nejlepší cesty v neúplném herním stromu pro hry dvou hráčů. Předpokládá, že hrající hráč chce z prověřovaného tahu získat maximum a zároveň chce, aby protivník tímto tahem získal minimum. Pro správné užití tohoto algoritmu je třeba dobře zvolit ohodnocující funkci, kterou ohodnotíme listy herního stromu. Zbytek uzlů směrem ke kořenu je ohodnocen tímto algoritmem. Na úrovni stromu MAX (hrající hráč) vybíráme maximum z následující úrovně, kdežto na úrovni MIN (protivník) vybíráme minimum z následující úrovně tak, jak je zobrazeno na příkladu na obrázku 2.1. Obrázky stromů hry byly vygenerovány z [Kraljic(2011)]. Pseudokód algoritmu má následující podobu:

```
function minimax(uzel, hloubka, hrac)
  if (uzel is list) or (hloubka <= 0) then
    return ohodnocujici_funkce
  if hrac = MaxHrac then
    for each potomek in uzel do
      hodnota := max(hodnota, minimax(potomek, hloubka-1,
                     not(hrac)))
    return hodnota
  else
    for each potomek in uzel do
      hodnota := min(hodnota, minimax(potomek, hloubka-1,
                     not(hrac)))
    return hodnota
```

Největší problém tohoto algoritmu je, že nám vrací dobré hodnoty pouze na několik tahů dopředu. Může se stát, že někdy v dalších tazích bude zjištěno, že celá dříve velice slibně rozvíjená větev je rázem nepoužitelná. To-

muto se říká *efekt horizontu*, neboť nejsme schopni vidět dále, než je hloubka stromu.



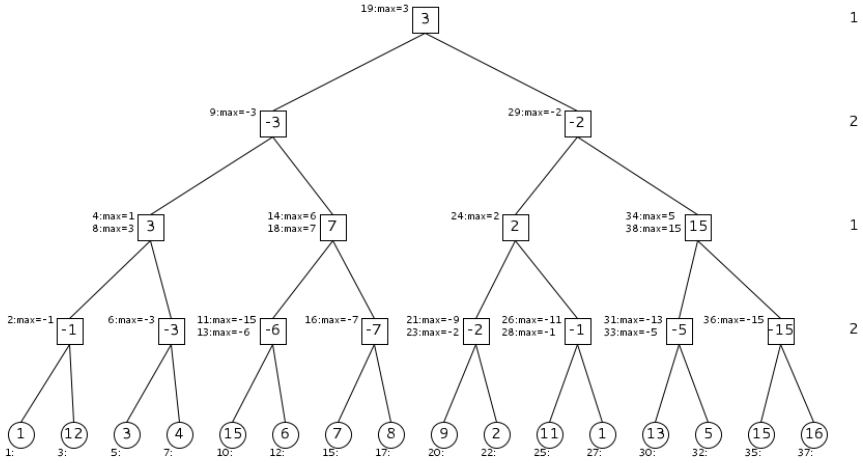
Obrázek 2.1: Herní strom po aplikaci algoritmu minimax. Kulaté prvky jsou ohodnocené listy stromu, hranaté uzly stromu. Čísla vedle listů a uzlů určují pořadí nalezení listu nebo změny uzlu spolu s hodnotou maxima a minima platné v daný okamžik.

2.2.2 Negamax

Negamax je elegantnější verze algoritmu minimax. Dává stejné výsledky za stejný počet iterací stromu. Výhodou je větší jednoduchost zápisu algoritmu. Namísto střídání vyhledávání maxima a minima zde hledáme ve vyšších úrovních pouze maximum. Do nižší úrovně ale následně přeneseme negaci nalezeného maxima. Příklad se nachází na obrázku 2.2. Pseudokód algoritmu má následující podobu:

```
function negamax(uzel, hloubka) : integer
    if (uzel is list) or (hloubka <= 0) then
        return ohodnocujici_funkce
    for each potomek in uzel do
        hodnota := max(hodnota, -negamax(potomek, hloubka-1))
    return hodnota
```

Nevýhoda v podobě efektu horizontu samozřejmě jako u algoritmu minimax zůstává.



Obrázek 2.2: Herní strom po aplikaci algoritmu negamax. Kulaté prvky jsou ohodnocené listy stromu, hranaté uzly stromu. Čísla vedle listů a uzlů určují pořadí nalezení listu nebo změny uzlu spolu s hodnotou maxima platnou v daný okamžik.

2.2.3 Alfa-Beta prořezávání

Alfa-Beta prořezávání opět není plnohodnotný algoritmus, ale pouze vylepšuje funkčnost minimaxu nebo negamaxu. Vylepšení spočívá v tom, že některé větve se mohou prohlásit za neperspektivní, a tudíž je možno je z dalšího prohledávání vynechat. Tímto snížíme počet iterací stromu a je ověřeno, že při použití alfa-beta prořezávání můžeme za přibližně stejný čas prohledat strom s dvojnásobnou hloubkou.

Neperspektivní větve zjistíme tak, že si na každé úrovni pamatujeme již nalezené maximum α, respektive minimum β (podle toho, na jaké úrovni jsme). Pokud z další větve tohoto uzlu přijde větší, respektive menší číslo, dále tuto větev prohledávat nebudeme, neboť tato větev již nižší úroveň nemůže ovlivnit. Následuje pseudokód pro alfa-beta prořezávání algoritmu minimax:

```
function minimaxAB(uzel, hloubka, alfa, beta, hrac)
  if (uzel is list) or (hloubka <= 0) then
    return ohodnocujici_funkce
  if hrac = MaxHrac then
    for each potomek in uzel do
      alfa := max(alfa, minimaxAB(potomek, hloubka-1, alfa,
                  beta, not(hrac)))
      if beta <= alfa then
        break
    return alfa
  else
    for each potomek in uzel do
      beta := min(beta, minimaxAB(potomek, hloubka-1, alfa,
                  beta, not(hrac)))
      if beta <= alfa then
        break
    return beta
```

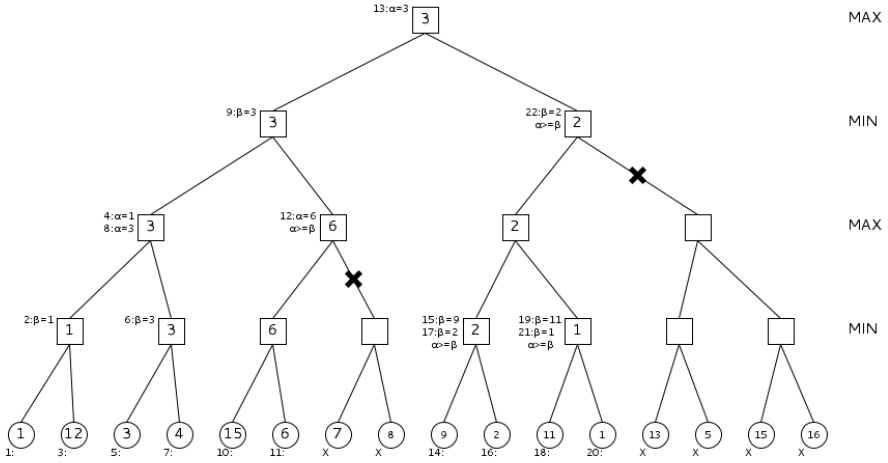
A pseudokód pro alfa-beta prořezávání algoritmu negamax:

```
function negamax(uzel, hloubka, alfa, beta) : integer
    if (uzel is list) or (hloubka <= 0) then
        return ohodnocujici_funkce
    for each potomek in uzel do
        hodnota := -negamax(potomek, hloubka-1, -beta, -alfa)
        if hodnota >= beta then
          return hodnota
        if hodnota >= alfa then
          alfa := hodnota
    return alfa
```

Příklad na algoritmus minimaxu s alfa-beta prořezáváním se stále stejným stromem je opět zobrazen na obrázku 2.3. Můžeme si všimnout, že tentokrát je řešení nalezeno již po 22 krocích. Alfa-beta prořezávání pro algoritmus negamax je řešen obdobně.

2.2.4 Strategie pro hry s více protihráči

Ve hře s více než dvěma hráči se také obvykle používá algoritmus minimax nebo negamax. Používá se ale tak, jako bychom hráli zároveň více her pro dva hráče s každým ze soupeřů zvlášť. Výsledný tah je pak zprůměrovaný



Obrázek 2.3: Herní strom po aplikaci algoritmu minimax s alfa-beta prořezáváním. Kulaté prvky jsou ohodnocené listy stromu, hranaté uzly stromu. Čísla vedle listů a uzlů určují pořadí nalezení listu nebo změny uzlu spolu s hodnotami α a β platnými v daný okamžik. Překřížené hrany značí větve, které byly vypuštěny.

výsledek ze všech těchto dílčích her.

2.3 Zvolení hry

Pro výběr hry jsem si zvolil kritérium, že by mělo jít o hru s úplnou informovaností, aby se daly využít výše popsané algoritmy. Taktéž bych tu hru měl znát, abych mohl lépe zhodnotit funkčnost a efektivnost algoritmů při hře s počítačem. Volba padla na hru Othello, která je možná více známa pod názvem Reversi. Jedná se o deskovou hru pro dva hráče.

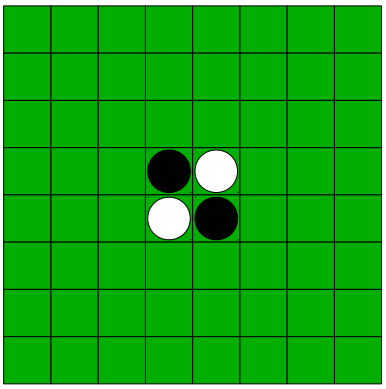
2.3.1 Pravidla hry Othello

Hra se hraje na ploše 8×8 čtverců s kulatými kameny, které jsou z jedné strany černé a z druhé strany bílé. Počáteční stav je uveden na obrázku 2.4. Tah začíná hráč s černými kameny. Kámen se musí na libovolné prázdné políčko umístit tak, aby mezi položeným kamenem a dalším vlastním kamenem na ploše v horizontálním, vertikálním, ale i diagonálním směru byl alespoň jeden kámen opačné barvy, tj. kámen protivníka. Tyto obestoupené kameny

protivníka se stanou vašimi, čili kameny se otočí a budou mít vaši barvu. V žádném směru nesmí být mezi obestoupenými kameny prázdné pole. Berou se kameny ze všech směrů, ve kterých jsou soupeřovi kameny obestoupeny. Nelze zajmout kameny, které se mezi dva vaše již položené kameny dostanou až v průběhu hry.

Pokud hráč nemůže žádný svůj kámen umístit, protože není možno zajmout žádný soupeřovo kámen, hraje dál soupeř. Pokud nemůže hrát ani soupeř, hra končí. Tato situace samozřejmě nastane v momentě, kdy je na ploše již všech 64 kamenů, ale může nastat i v průběhu hry. Důležité pravidlo je, že pokud hráč může hrát, pak hrát musí. Nelze se tahu vzdát.

Vyhrává hráč, který má na konci hry na ploše více kamenů. Za každý kámen má hráč jeden bod. Pokud hra skončí dříve, než jsou všechny kameny na ploše, pak si vítěz připočte i všechna volná místa. Součet bodů obou soupeřů je tedy vždy 64. Pravidla byla volně převzata z [Klu(2006)].



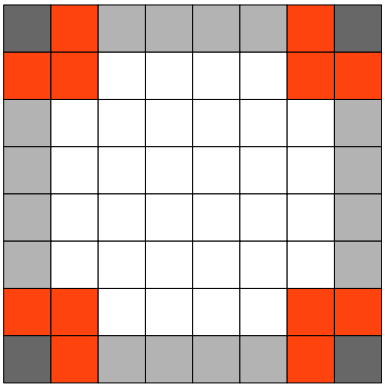
Obrázek 2.4: Plocha a počáteční stav hry Othello.

2.4 Analýza implementace hry Othello

Úplný strom pro tuto hru vygenerovat nebude možné, protože ačkoliv hra může mít maximálně 60 tahů, strom s hloubkou 60 by stejně obsahoval 10^{58} uzlů. Takže se přímo nabízí použití algoritmu minimax, resp. jeho elegantnějšího řešení negamax. Samozřejmě včetně alfa-beta prořezávání, abychom mohli prohledávat do větší hloubky.

2.4.1 Ohodnocující funkce

Problém nastane při ohodnocení listů stromu. Je naivní si myslet, že by stačila funkce hodnotící podle počtu získaných nebo ztracených kamenů. I z vlastní zkušenosti vím, že tato strategie je naprosto nefunkční. Mnohem důležitější je dávat si pozor na určitá pole ve hře, nebo se naopak na jiná snažit dostat. Velice strategická místa jsou rohy hrací plochy, neboť není možné žádným legálním tahem tyto kameny zajmout. O něco méně strategické jsou pak hrany hrací plochy, kde je menší možnost zajmutí. Naopak nestrategické jsou pole, díky kterým se protivník může dostat během dalších tahů do rohů. Grafické zobrazení těchto polí je na obrázku 2.5.



Obrázek 2.5: Strategické pozice ve hře. Nejvýhodnější pozice jsou tmavě šedá pole, dále pak světle šedá. Bílá jsou neutrální bezpečná pole. Červená jsou nevýhodná pole.

Dalším problémem během hry může být situace, kdy nebudeme schopni zahrát legální tah a bude hrát znovu protivník. Abychom se tomuto vyhnuli, mohli bychom jako další kritérium zvolit počet legálních tahů. Výsledná funkce by pak byla součtem výše popsaných kritériích, kdy každé by mělo mít jinou váhu, neboť nejsou stejně důležitá. Funkce by mohla vypadat takto:

$$f = 100 \times S_{pozice} + 10 \times S_{\ tahy} + S_{\ kameny}$$

Tato funkce nám říká, že si ceníme desetkrát více informace o uskutečnitelných tazích a ještě desetkrát více vhodného umístění.

2.4.2 Škálovatelnost

Jak je ve hrách obvyklé, i zde by bylo dobré mít možnost nastavit obtížnost počítačového protivníka. Jednou z možností by mohla být změna ohodnocující funkce, kdy například snížíme váhu jednotlivých složek nebo některé úplně vynecháme.

Lepší možnost ale bude upravit algoritmus vyhledávání v herním stromu tak, aby generoval hloubku stromu přímo úměrnou nastavené obtížnosti (čím těžší obtížnost, tím hlubší herní strom).

2.5 Nároky na aplikaci

Na aplikaci Othello bychom měli vznést určité požadavky. Předně by se mělo jednat o grafickou aplikaci, díky které by hraní hry vizuálně co nejvíce odpovídalo realitě. Ovládání by mělo být primárně myší, u tohoto typu deskových her je nejjednodušší. Aplikace by mohla být multiplatformní a vícevláknová, minimálně k oddělení výpočetního jádra od uživatelské a systémové obsluhy aplikace. Taktéž by mohla umožnit hrát s jiným hráčem prostřednictvím sítě.

Programovací jazyk je možné zvolit jakýkoliv. Méně vhodné jsou různé skriptovací jazyky jako Perl nebo Python, neboť aplikace bude využívat intenzivně výpočetní výkon a tyto jazyky mají zbytečnou režii. Vhodné jsou klasické jazyky jako C++ a Java.

2.6 Možnosti hraní po síti

Jelikož komunikace po sítí není cílem této práce, ale jen doplněk, budu se jí zabývat jen velice okrajově.

Síťová komunikace se dělí na vrstvy. Každá vrstva se stará o jinou činnost na síti prostřednictvím protokolů. Protokol je sada určitých pravidel, algoritmů a jiných mechanismů, které zajišťují efektivní komunikaci hardware a software na síti. Nejznámější sada (rodina) protokolů je TCP/IP. Tyto protokoly jsou základním kamenem celosvětové sítě internet a postupně začaly pronikat i do lokálních a podnikových sítí. Dnes už je TCP/IP v podstatě standard síťové komunikace.

Protokoly TCP/IP se dělí do čtyř vrstev:

1. **aplikační vrstva** (nejvyšší vrstva) - tvoří ji služby pro koncového uživatele
2. **transportní vrstva** - zajišťuje přenos dat mezi dvěma body na síti
3. **síťová vrstva** - zajišťuje adresaci, směrování a předávání dat (IP protokol)
4. **vrstva síťového rozhraní** (nejnižší vrstva) - zajišťuje přístup k hardware a v řadě sítích není implementována (samotná síť se pak stará o tuto vrstvu)

Jelikož cílem bude pouze zajistit přenos dat (tahů hráčů) mezi počítači, nebudu se již dále síťovou vrstvou a vrstvou síťového rozhraní zabývat. V této kapitole bylo čerpáno z [Kozierok(2005)].

2.6.1 Transportní vrstva

V transportní vrstvě jsou k dispozici pro přenos dat dva protokoly:

TCP Spolehlivý protokol přenosu dat. Data vždy doručí a ve správném pořadí. Je spojovaný se třemi fázemi: spojení, přenos, rozpojení. Pro rozeznání příjemce používá porty, což jsou bez-znaménková 16 bitová čísla. Nevýhodou je nižší rychlost přenosu z důvodu zajištění všech popsaných vlastností.

UDP V podstatě opak TCP. Data nemusí doručit nebo je nemusí doručit ve správném pořadí. Díky tomu je ale rychlejší než TCP. Má pouze fázi přenosu. Porty pro identifikaci aplikace používá také.

2.6.2 Aplikační vrstva

Aplikační vrstva obsahuje protokoly, jako například HTTP, FTP, DNS, Telnet a jiné. Tyto protokoly již rovnou říkají, jaký typ dat přenáší. Pro potřeby mojí aplikace ale žádný z těchto protokolů nebude vhodný. Jsou zbytečně složité. Namísto toho velice jednoduchý navrhnu sám.

2.6.3 Komunikační modely

Komunikační model mezi dvěma počítači již žádná vrstva TCP/IP neřeší. Existují dva základní:

Klient/Server Dva počítače (klienti) jsou mezi sebou propojeni skrze prostředníka (server). Server slouží pouze k vyřizování požadavků klientů. Může si vést seznam připojených klientů a ten pak sdílet mezi klienty. Klient pak pro připojení k jinému klientu nemusí znát jeho adresu. Veškerá komunikace v Klient/Serverovém modelu jde přes server. Pokud bude mít server veřejnou adresu, je možné přes něj spojit i dva klienty s neveřejnými adresami.

Peer-To-Peer Dva počítače jsou spojené přímo mezi sebou. Oba vystupují zároveň jako klient i server. Nelze spojit dva počítače s neveřejnými adresami v oddělených sítích. Výhoda je právě v nepotřebnosti serveru.

3 Realizace aplikace Othello

3.1 Vybraná řešení k implementaci

Jak jsem již naznačil v kapitole 2.4, pro strategii hry počítače jsem vybral algoritmus negamax s alfa-beta prořezáváním. Ohodnocující funkci jsem nakonec ještě rozšířil a její podoba je následující:

$$f = V + P + 10.000 \times S_{roh} + 100 \times S_{\ okraj} + S_{\ pole}$$

$V = 1.000.000$ - pokud daný stav je výhra hráče
$P = -100.000$ - pokud hráč nemůže zahrát tah
S - počet kamenů hráče v rozích (roh), na okrajích (okraj), kdekoliv jinde (pole)

Dále jsem zvolil tři obtížnosti počítačového hráče:

- lehká - počítač zahrává tahy náhodně
- střední - negamax s hloubkou prohledávání 2
- těžká - negamax s hloubkou prohledávání 5

Pro síťovou komunikaci jsem zvolil protokol TCP, protože naprosto nezáleží na rychlosti přenosu. Budou se přenášet zanedbatelné velikosti dat, ale je potřeba je doručit v pořádku. Vzhledem k tomu, že Othello je striktně hra pro dva hráče, jako síťový model jsem zvolil Peer-To-Peer. V aplikačním protokolu se přenášejí řetězce znaků. První znak je příkaz, další jsou parametry. Popis příkazů je v následující tabulce:

příkaz	1. parametr	2. parametr	popis
C	číslo hráče		žádost o připojení od hráče hrajícího jako černý nebo bílý
M	sloupec	řádek	zahrán tah na souřadnice [sloupec, řádek]

3.2 Vývojové prostředí

Pro realizaci programu jsem se nakonec rozhodl využít můj oblíbený jazyk Pascal. Jeho otevřený překladač FPC (Free Pascal Compiler) je na velice

dobré úrovni a generuje bez problémů multiplatformní kód pro mnoho různých architektur. Ruku v ruce s FPC jde projekt Lazarus, což je IDE vycházející z nejpopulárnějších Delphi 7. Kromě IDE obsahuje i knihovnu LCL, což je ekvivalent knihovny VCL u Delphi. Stabilita IDE je dnes již dobrá a vytváření GUI aplikací je stejně jednoduché, jako bylo v prostředí Delphi. Výhodou je, že výsledný spustitelný soubor nevyžaduje žádné další knihovny ani běhové prostředí (run-time environment). Nevýhodou je, že většina nutných věcí je sestavena do tohoto souboru, a díky tomu i velice jednoduché programy mají velikost v řádu megabytů. V praxi to ale má pouze minimální dopad na dobu spouštění aplikace.

Dále jsem použil následující komponenty:

BGRABitmap Nevizuální komponenta pro rychlou manipulaci s obrázky a s podporou alfa kanálu na všech systémech (včetně Linux GTK2, kde podpora alfa-kanálu v LCL chybí). Jedná se o nezávislou bitmapu uloženou v paměti, na kterou můžeme kreslit. Po skončení kreslení se celá tato bitmapa zkopíruje na plátno libovolné grafické komponenty (např. formuláře), čímž ji zobrazíme.

BGRAControls Vizuální komponenty založené na komponentě BGRABitmap. Vznikly primárně k nahrazení klasických prvků s podporou obrázků, které nepodporují alfa-kanál, za prvky, které používají BGRABitmap, a tudíž alfa-kanál podporují. Z těchto komponent používám BGRAVirtualScreen. Jedná se o plátno pro BGRABitmapu, které se samo stará o překreslení v nutných případech (např. změna velikosti okna).

lNet lNet neboli Lightweight Network Library je, jak název napovídá, jednoduchá síťová komponenta pro práci s protokolem TCP nebo UDP a s řadou aplikačních protokolů jako jsou HTTP a FTP. Existují i mnohem robustnější řešení, jako např. komponenta Synapse, ale pro účely této práce je komponenta lNet vhodná.

3.3 Jednotky (units)

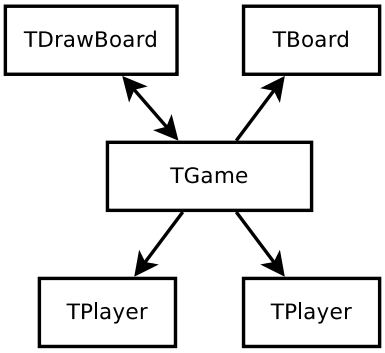
Každá mnou vytvořená jednotka má název `Unit*` a je uložena v souboru se stejným názvem `unit*.pas`. Pro lepší přenositelnost mezi operačními systémy obsahují všechny názvy zdrojových souborů pouze malá písmena. Po-

kud je v jednotce implementován formulář, je její název `UnitForm*`, a je proto uložena v souboru `unitform*.pas`. Jednotlivé jednotky nebudu popisovat, jelikož každá striktně obsahuje pouze definici a implementaci jedné třídy. Ty budou popsány v další kapitole. To znamená, že při zmínce např. o třídě `TGame` nalezneme její zdrojový kód v jednotce `UnitGame` v souboru `unitgame.pas`.

Jediná výjimka je jednotka `UnitCommon`, která neobsahuje žádnou třídu, ale pouze globální konstanty, proměnné, definice uživatelských struktur a funkce společné pro všechny ostatní jednotky.

3.4 Třídy (classes)

Vytvořené třídy jsou dále popsány pouze orientačně. Mnohem obšírněji jsou okomentovány přímo ve zdrojových kódech v příloze B nebo na přiloženém CD. Jednoduchý diagram se závislostmi vytvořených tříd je na obrázku 3.1.



Obrázek 3.1: Závislosti vytvořených tříd. `TPlayer` má dvě instance, pro každého hráče jednu.

3.4.1 Třída TGame

Tato třída se stará o samotný průběh hry. Vytváří si instanci třídy `TBoard` (proměnná `FBoard`). To je pole hrací desky, na kterém probíhá hra. V metodě `Play` je obsažena smyčka, ve které se čeká na tahy obou hráčů až do konce hry, nebo dokud hra není přerušena zvnějšku (např. ukončení aplikace). Dále zjišťuje možné tahy, legálnost tahů, vytváří historii tahů a na vyžádání vrací tahy zpět. V globální proměnné `Game` se vytvoří na začátku hry instance této třídy a zruší se na konci hry.

3.4.2 Třída TDrawBoard

Tato třída byla původně součástí třídy `TBoard`. K jejímu odtrhnutí došlo v momentě, kdy se původní třída začala používat ve vyhledávacím algoritmu počítačového hráče. Tam nebylo potřeba mít grafické funkce, protože nám šlo pouze o vyhledání nejlepšího tahu, nikoliv o jeho vizualizaci.

Jak už vyplývá z prvního odstavce, tato třída obsahuje metody a funkce potřebné k vizualizaci herní desky `Game.FBoard`. Metoda `Draw` je volána hlavním oknem, pokud se například změní velikost okna. Tato metoda spočítá okraje a velikost políček. Všechna políčka nakonec vykreslí do komponenty `BGRAVirtualScreen` v hlavním okně. Dále obsahuje konverzní funkce mezi souřadnicemi `BGRAVirtualScreen` a souřadnicemi hrací plochy.

Globální proměnná `DrawBoard` obsahuje instanci této třídy a je vytvořena při spuštění aplikace.

3.4.3 Třída TBoard

Tato třída obsahuje pole hrací desky `FBoard` a metody pro manipulaci s ním. Metoda `DoMove` provede tah včetně zajmutí všech soupeřových kamenů. `FindPossibleMoves` zase vrátí pole možných tahů.

3.4.4 Třída TPlayer

Třída `TPlayer` reprezentuje obecného hráče. Slouží jako mateřská třída pro implementace různých typů hráčů, které z této třídy dědí základní metody volané z třídy `TGame`. Těmito metodami jsou:

Init je volaná pouze před začátkem hry

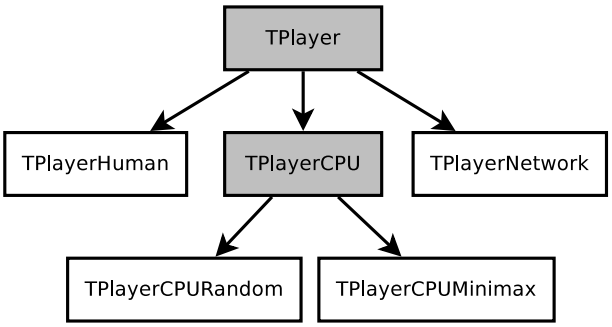
MoveBegin je volaná vždy před začátkem tahu

Moving čeká na tah hráče a tento tah vrací jako výsledek

MoveEnd je volaná vždy na konci tahu

Deinit je volaná na konci hry

Na obrázku 3.2 je diagram tříd, které jsou zděděné z této třídy. Pouze třídy v bílém obdélníku jsou používány jako instance hráčů.



Obrázek 3.2: Třída `TPlayer` a její dědicové.

3.4.5 Třída TPlayerHuman

Reprezentuje hráče, který je ovládán člověkem za použití myši. Implementuje tyto metody z `TPlayer`:

MoveBegin nastaví metody pro zpracování pohybu myši a stisku tlačítek myši

Moving čeká, až uživatel vybere tah za pomocí myši, a tento vrátí

MoveEnd přestane zpracovávat události myši

Navíc obsahuje metodu `MouseMove`. Ta nám zvýrazňuje políčko hrací desky, nad kterým se právě nachází kurzor myši.

3.4.6 Třída TPlayerCPU

Toto je mateřská třída pro všechny hráče ovládané počítačem. Vznikla proto, abychom mohli počítačový tah zpomalit na hodnotu z nastavení aplikace. Tím dosáhneme lepší přehlednosti hry, zejména při hraní dvou počítačů proti sobě. Implementuje tyto metody:

MoveBegin aktivuje časovač a spustí odpočet

MoveEnd čeká, až odpočet doběhne, a deaktivuje časovač

Pokud výpočet tahu v potomcích v metodě `Moving` bude trvat déle než je hodnota časovače, pak se samozřejmě v `MoveEnd` na nic nečeká.

3.4.7 Třída TPlayerCPURandom

Nejjednodušší implementace počítačového protivníka. V metodě `Moving` vrací pouze náhodný tah.

3.4.8 Třída TPlayerCPUMinimax

Počítačový protivník, který pro výpočet optimálního tahu používá algoritmus *negamax* s *alfa-beta prořezáváním*. Nachází se v metodě `Minimax`. Vytváří si vlastní instanci `TBoard` v proměnné `Board`, nad kterou algoritmus *negamax* probíhá. Implementuje tyto metody:

MoveBegin zkopíruje stav hrací desky do `Board`

Moving zjistí možné tahy a každý ocení algoritmem *negamax*, nejlepší pak vrátí

Dále je zde metoda `Winner`, která určí vítěze a `Evaluation`, což je ohodnocující funkce algoritmu *negamax*.

Tuto třídu jsem původně zamýšlel udělat vícevláknovou. Podařilo se vytvořit metodu `Minimax` uvnitř samostatného vlákna. S funkčností tohoto řešení jsem ale nebyl spokojen. Tah byl nalezen až za dvojnásobné množství času než při použití klasického řešení. Jádro procesoru nebylo při výpočtu ani plně vytíženo. Důvodem bylo použití metody `TThread.WaitFor`, která čeká na dokončení běhu vlákna. Tato metoda ale nemá být volána z hlavního vlákna, ve kterém běží i aplikační smyčka. To byl bohužel můj případ. Řešení tohoto problému by znamenalo přepsat valnou část kódu a všechny metody, které čekají na nějaký výsledek, nahradit událostmi. Tím bych nemusel použít metodu `WaitFor`, protože těsně před skončením běhu vlákna by toto vlákno vyvolalo událost. Tuto událost by pak obsloužila příslušná třída čekající na výsledek.

Z toho důvodu jsem musel do metody `Minimax` přidat volání metody `Application.ProcessMessages`. To zajistí provedení všech událostí v aplikaci, jako je obsluha myši a oken. V opačné případě by bylo okno aplikace po dobu výpočtu ve stavu „neodpovídá“, což je z uživatelského hlediska nepřípustné. Pokud ale bylo toto volání uskutečněno v každé iteraci metody, výpočet se opět neúnosně zpomalil. Proto je volání uskutečněno každou stou iteraci. To se zdá být dobrým poměrem mezi časem výpočtu a odezvou okna na požadavky uživatele.

3.4.9 Třída TPlayerNetwork

Jedná se o implementaci síťového hráče. Vytvoří si komponentu `TlTcp` z balíku lNet pro síťovou komunikaci protokolem TCP. Implementuje tyto metody:

Init zobrazí formulář připojení a snaží se vytvořit spojení

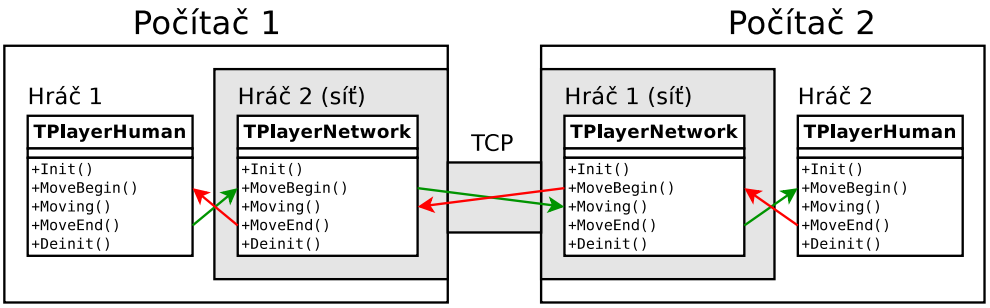
MoveBegin pošle po síti poslední tah soupeře

Moving čeká na tah poslaný po síti a vrátí ho

Deinit zavře spojení

Dále obsahuje metody pro zpracování událostí komponenty `TlTcp`: `OnError`, `OnDisconnect` a `OnReceive`.

Schéma hraní hry po síti je vidět na obrázku 3.3.



Obrázek 3.3: Schéma hraní hry po síti. Zelenými šipkami je znázorněn tah prvního hráče, červenými druhého hráče.

3.5 Testování hry

Hru jsem testoval hlavně sám proti počítači. Po nějaké době jsem si všiml, že poměrně hodně vyhrávám i s těžkým počítačem. Někdy počítač zahrál vyloženě špatný tah. Do algoritmu *negamax* jsem tedy zanesl mnoho ladicích výpisů a nalezl jsem fatální chybu, která vznikla již ve fázi analýzy. Původně jsem měl v třídě `TBoard` kromě `DoMove(TMove)` ještě metodu `UndoMove(TMove)`, která reverzním algoritmem odebrala tah `TMove` a vrátila i všechny zajaté kameny soupeři. A právě zde byl problém. `UndoMove` v určitých stavech vrátila soupeři více kamenů, než kolik `DoMove` zajala. Na tyto

stavy jsem v analýze nepomyslel a došel k názoru, že pouze zapamatovanými tahy jsem schopen vrátit stav hry na jakékoliv místo v historii tahů. Toto však nebyla pravda. Do historie tahů jsem musel kromě tahu ještě přidat kopii pole herní desky příslušném okamžiku. Pak se již výhra nad počítačem pro mne stala otázkou náhody.

Dále jsem testoval můj program proti jiným programům. Testoval jsem to tím způsobem, že jsem já byl v podstatě „TCP protokol“ mezi těmito programy.

jméno programu nebo adresa apletu	výsledek
http://www.gamesforthebrain.com/game/reversi/	2:1
http://www.fetchfido.co.uk/games/reversi/reversi.htm	ne[a]
http://www.games.com/game-play/reversi/single/	0:3
KReversi	0:3
Iagno	ne[b]
http://www.mathsisfun.com/games/reversi.html	2:1
http://www.artifactinteractive.com.au/arcade/reversi.html	2:1
WZebra	0:3
Daisy Reversi	3:0
http://www.ireversi.com/	3:0

[a] špatný algoritmus, kdykoliv nemohl zahrát tah, skončil hru s tím že vyhrál
[b] program přestane reagovat, když nemůže počítač zahrát tah

Použil jsem náhodným výběrem implementace Othella či Reversi psané ve Flashi nebo JavaScriptu na webových stránkách a stáhl i pár programů. Zatímco s webovými implementacemi Othella neměl můj program větší problémy, se staženými programy spíše prohrával. Proto bych svůj algoritmus (lépe řečeno ohodnocující funkci) hodnotil jako průměrný.

Dále mne zajímalo, jaký vliv má alfa-beta prořezávání v algoritmu negamax (minimax) na výsledný čas prohledávání. Dal jsem proti sobě hrát dva počítače se stejnou hloubkou prohledávání stromu s alfa-beta prořezáváním i bez něj. Navíc jsem musel vypnout možnost náhodného výběru nejlepších tahů. Tím jsem docílil toho, že každá hra se stejnou hloubkou měla totožné tahy. Měřil jsem čas od začátku hry do jejího konce.

Naměřené hodnoty jsou pouze orientační, protože s jinou hloubkou stromu se mění i odehraná hra, která může trvat odlišnou dobu. Přesto tabulka 3.1 ukazuje, že přidání čtyř řádek kódu (alfa-beta prořezávání) se bohatě vyplatí, neboť za stejný čas jsme schopni prohledat o přibližně třetinu hlubší strom.

hloubka	bez alfa-beta	s alfa-beta
9		12m 7s
8		58,1s
7		24,7s
6	7m 32s	6,6s
5	46,1s	2,1s
4	2,4s	0,3s
3	0,3s	0,1s
2	0,1s	0,1s

Tabulka 3.1: Naměřené hodnoty doby trvání jedné hry s použitím alfa-beta prořezávání a bez něho.

3.6 Požadavky na spuštění aplikace

Binární spustitelný soubor byl vytvořen pro následující platformy (v závorkách jsou systémy, ve kterých byla aplikace úspěšně testována):

- Win 32-bit (Windows XP, Windows 7 64-bit)
- Linux GTK2 32-bit (Ubuntu 32-bit)
- Linux GTK2 64-bit (Ubuntu 64-bit, Arch Linux 64-bit)
- Mac OSX Carbon 32-bit (Lion 10.7.3 64-bit)

Aplikace zabírá v paměti necelých 10 MB. Požadavky na hardware nejsou vyšší než pro daný systém. Aplikace nakonec není vícevláknová, takže postačí jednojádrový procesor. Není potřeba ani grafická karta s akcelerací 3D grafiky. Na HDD je třeba mít pouze 10 MB volného místa, ale aplikace lze bez problémů spouštět i z CD.

V Linuxu lze spustit pouze v libovolném grafickém prostředí. Pro svůj běh používá GTK2 widgeset. Z toho plyne, že je nativně vytvořená pro grafické prostředí Gnome 2. V prostředí Unity (Ubuntu) podporuje globální menu.

3.7 Překlad

3.7.1 Instalace IDE Lazarus

Pro systém Windows je instalační soubor umístěn na přiloženém CD ve složce `install\lazarus\win32`. Nainstaluje se i překladač FPC.

Ve většině Linuxových distribucích by mělo stačit nainstalovat balíček *lazarus* z oficiálních repozitářů. Balíčkovací systém by se měl vypořádat se všemi ostatními závislostmi (včetně FPC) automaticky.

Instalační balíčky DMG pro systém Mac OS X jsou na CD ve složce `install/lazarus/osx`. Je třeba nainstalovat všechny tři balíčky. Dále je třeba mít nainstalované Xcode s podporou *Command line tools*. V Xcode 4 se instaluje tato podpora přímo z aplikace v sekci *Preferences / Download*. V Xcode 3 stačí tuto podporu zaškrtnout v možnostech při instalaci.

3.7.2 Instalace komponent

Do IDE Lazarus musíme doinstalovat balíčky *bgrabitmap*, *bgracontrols* a *lnetvisual*. Tyto se nacházejí na CD ve složce `install\components`. Instalují se přímo v prostředí Lazarus volbou *Package / Open package file (.lpk)*. Otevřeme soubor s příponou `.lpk` nacházející se v příslušných složkách komponent. Pak již stačí kliknout na tlačítko *Install* a zvolit možnost *Rebuild Lazarus*. Toto provedeme pro všechny balíčky.

Lazarus po instalaci balíčků skutečně potřebuje přeložit sám sebe. Proto je potřeba, aby do složek, kde je Lazarus nainstalován, měl uživatel právo zápisu.

3.7.3 Překlad aplikace

V IDE Lazarus stačí otevřít projekt Othello volbou *Project / Open project* a nalézt soubor `othello.lpi` ve složce se zdrojovými kódy. Soubory se zdrojovými kódy se nacházejí na CD ve složce `src`. Pak je již možné zvolit volbu *Run / Run* pro překlad a spuštění aplikace Othello nebo *Run / Build* pro překlad bez spuštění. Do složky `src` je opět třeba mít právo zápisu, proto nelze překládat projekt přímo z CD.

4 Závěr

Cílem této bakalářské práce bylo prozkoumat možnosti herních strategií a následně vytvořit programový systém umožňující hrát vhodnou hru. Herní strategie jsou implementovány algoritmy, které procházejí neúplný herní strom. Ten představuje všechny možné tahy na několik kol dopředu. Strategie spočívá ve vyhledání nejlepšího tahu z tohoto stromu.

Vhodnou hrou byla zvolena desková hra pro dva hráče Othello. Vytvořil jsem multiplatformní aplikaci v jazyku ObjectPascal, do níž jsem implementoval celkem tři typy hráčů:

- člověk hrající prostřednictvím myši
- počítač vybírající tahy náhodně nebo jako výsledek zvoleného algoritmu negamax
- síťový hráč hrající na jiném počítači propojeném pomocí TCP protokolu

Všechny typy hráčů lze libovolně kombinovat (vyjma dvou hráčů po síti).

Algoritmus negamax byl zvolen kvůli přehlednějšímu zápisu v porovnání s algoritmem minimax. Přidání alfa-beta prořezávání do tohoto algoritmu umožnilo prohledat za stejný čas větší hloubku stromu, neboli možné tahy více kol dopředu.

Nepodařilo se však tento algoritmus udělat vícevláknový z důvodu pro tento účel nevhodně zvolené struktury aplikace. Vzhledem ke skutečnosti, že procházení stromem je ideální úlohou pro vícevláknové použití, vidím v tomto případnou možnost vylepšení aplikace.

Literatura

[Klu(2006)] *Česká federace Othello*. Klub deskových her Paluba, 2006. Dostupné z: `http://othello.hrejsi.cz/`.

[Kozierok(2005)] KOZIEROK, C. M. *The TCP/IP Guide: A Comprehensive, Illustrated Internet Protocols Reference*. : No Starch Press, 2005. ISBN 978-1593270476.

[Kraljic(2011)] KRALJIC, K. *Game Visualization*, 2011. Dostupné z: `http://ksquared.de/gamevisual/launch.php`.

[Russell(1995)] RUSSELL, S. J. *Artificial Intelligence: A Modern Approach*. Englewood Cliffs, New Jersey 07632 : Prentice Hall, 1995. ISBN 0-13-103805-2.

[Vopravil(2007)] VOPRAVIL, V. *Úvod do teorie kombinatorických her*, 2007. Dostupné z: `http://cgt.ic.cz/cgt_.html`.

A Uživatelská příručka

A.1 Spuštění a první hra

Celá aplikace sestává pouze z jednoho spustitelného souboru `othello.exe` (Windows), `othello` (Linux) nebo `othello.app` (OSX). Nepoužívá žádné další soubory, ani žádné soubory nikde nevytváří. Po spuštění se zobrazí okno s prázdnou hrací plochou. Pokud chceme hned hrát, pak z menu *Hra* vybereme možnost *Nová hra* a v nově otevřeném okně zvolíme, kdo bude hrát za jakou barvu. Pokud budeme chtít hrát proti počítači a začínat, ponecháme u černého hráče výběr člověk a u bílého hráče vybereme jednu ze tří obtížností počítače. Klikneme na *Začít hru*.

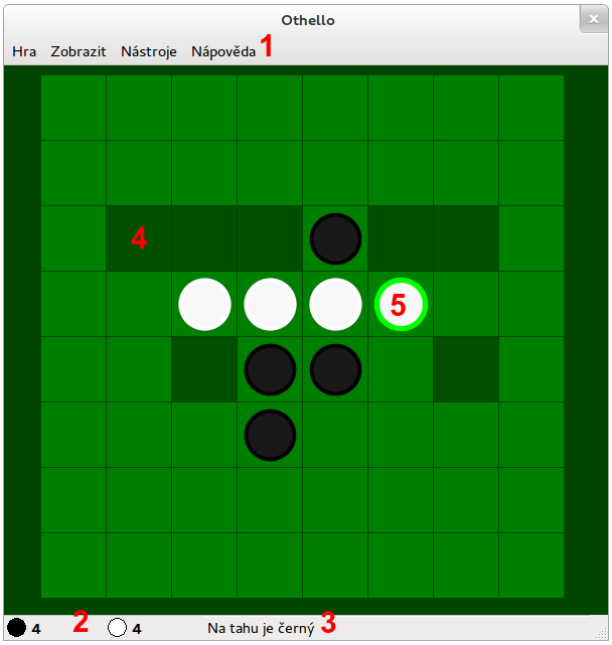
Hra se ovládá myší. Stačí kliknout levým tlačítkem na políčko, kam chceme umístit kámen. Pokud se kámen neobjeví, pak tento tah není legální a musíme zvolit jiný legální tah. Pro lepší orientaci můžeme zaškrtnout v menu *Zobrazit* položku *Možné tahy*. Díky tomu se budou zvýrazňovat políčka, kam lze položit kámen právě hrajícího hráče. V menu *Zobrazit* lze dále zaškrtnout volbu *Poslední tah*. Ta nám navíc zvýrazní naposledy položený kámen, ať už náš, nebo soupeřův.

Po skončení hry se zobrazí okno s výsledkem a možností hrát stejnou hru ještě jednou (tlačítko *Opakovat hru*), nebo zvolit jiné hráče (tlačítko *Nová hra*). Aplikace se ukončuje standardní cestou zavřením okna nebo z menu *Hra* volbou možnosti *Konec*.

A.2 Hlavní okno

Hlavní okno aplikace je na obrázku A.1. Obsahuje tyto elementy:

1. menu aplikace (dosud nezmiňované položky popsány níže)
2. ukazatel aktuálního skóre obou hráčů
3. výpis stavu hry (kdo je na tahu nebo že hráč nemohl hrát)
4. zvýrazněné pole, kam lze umístit kámen
5. zvýrazněný naposledy položený kámen



Obrázek A.1: Hlavní okno aplikace Othello s právě probíhající hrou a červeně očíslovanými elementy.

Během hry lze vrátit náš poslední tah. To provedeme volbou *Tah zpět* v menu *Nástroje*. V praxi se samozřejmě napřed vrátí soupeřův tah, pak ten náš. Touto volbou opakovaně můžeme vrátit hru až do výchozí pozice. Tato volba není aktivní, pokud hrajeme síťovou hru.

Dále je možné zobrazit informace o aplikaci z menu *Nápověda / O programu*. Poslední nezmiňovanou volbou jsou nastavení aplikace v menu *Nástroje / Nastavení*. Tyto jsou vysvětleny dále v samostatné kapitole.

A.3 Síťová hra

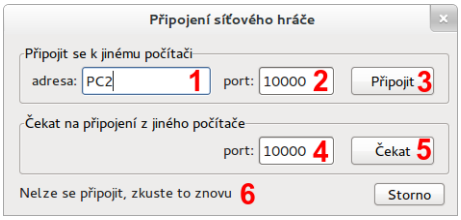
Pokud chceme hrát Othello po síti, je třeba nastavit jako jednoho hráče síť v okně *Nová hra*. Oba hráče jako síť nelze nastavit. Po kliknutí na *Začít hru* se nám zobrazí okno z obrázku A.2. Máme dvě možnosti připojení. Buď budeme čekat na připojení druhého hráče nebo se sami budeme snažit na druhého hráče připojit. Druhý hráč musí vždy zvolit opačnou možnost, jinak ke spojení nedojde. Ke spojení taktéž nedojde, pokud v době pokusu o připojení připojovaný hráč ještě nečekal na spojení.

A.3.1 Čekání na spojení

V textovém poli, označeným červeným číslem 4 na obrázku A.2 (dále již budu uvádět pouze čísla v závorce), nastavíme číslo TCP portu, na kterém budeme očekávat spojení. Výchozí port 10000 není potřeba měnit. Klikneme na tlačítko *Čekat* (5) a aplikace nám vypíše stav na místě (6). Pokud je port volný, aplikace vypíše, že čeká na tomto portu. Pokud ale port volný není, vypíše na místě (6) chybu. Pak je nutné port (4) změnit na jiné číslo.

A.3.2 Připojit se k jinému počítači

V textovém poli (1) zadáme buďto název počítače nebo IP adresu počítače připojovaného hráče. V poli (2) pak TCP port na kterém počítač připojovaného hráče čeká na spojení. Tlačítkem *Připojit* (3) se pokusíme o spojení. Pokud se připojení nezdaří, je nám o tom podána zpráva na místě (6). Pokud se spojení zdaří, je ještě možnost jedné chyby: oba hráči se snaží hrát za stejnou barvu. V tomto případě se opět vypíše zpráva na místě (6) a spojení se ukončí. Je třeba tedy aby jeden z hráčů změnil barvu kamenů. Nejrychleji to provede tak, že klikneme na tlačítko *Storno* a v předcházejícím okně *Nová hra* klikneme na malé tlačítko uprostřed okna <>, které slouží k prohození typu hráčů mezi barvami. Opět klikneme na *Začít hru* připojení by již mělo proběhnout v pořádku. To se projeví tak, že okno *Připojení síťového hráče* se samo zavře a hra začne. Hra probíhá stejně jako s jiným hráčem, jen nelze vracet tahy zpátky.



Obrázek A.2: Okno pro připojení hráče po síti s červeně vyznačenými elementy.

A.3.3 Nastavení aplikace

Z menu *Nástroje / Nastavení* vyvoláme okno, kde můžeme upravit chování programu. Okno můžeme vidět na obrázku A.3 včetně červeně vyznačených

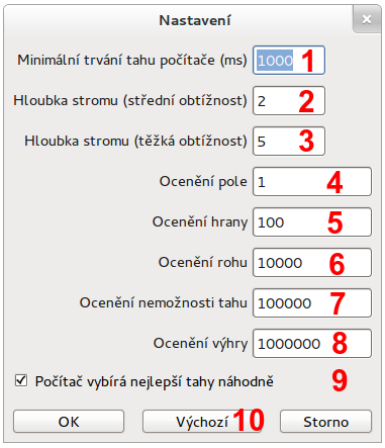
elementů (1) až (10). *Minimální trvání tahu počítače* (1) je minimální doba v milisekundách za kterou počítač odehraje tah. Tah ale může trvat i déle, záleží na rychlosti výpočtu tahu počítačem. Výchozí hodnota je 1000 ms, čili jedna sekunda. Toto nastavení se projeví i během hry.

V nastavení *Hloubky stromu* (2) a (3) určíme, kolik tahů dopředu bude počítač za danou obtížnost promýšlet. Pozor na zvolené číslo. Pokud bude moc velké, stěží se kdy dočkáte od počítače tahu. V tomto případě je potřeba hloubku stromu změnit na nižší číslo a hru spustit znovu.

Volby *Ocenění* (4) až (8) slouží k výpočtu nejlepšího tahu počítače. Čím větší číslo, tím daná pozice nebo stav hry má pro počítač větší váhu. Těmito nastaveními můžeme kompletně změnit chování počítače ve hře.

Počítač vybírá nejlepší tahy náhodně (9) je nastavení, kdy zajistíme (pokud je políčko zaškrtlé), že počítač nebude hrát stále stejně. Pokud totiž bude mít na výběr z více stejně dobrých tahů, vybere si jedno z nich náhodně. V opačné případě vybere vždy první z nich.

Pokud budeme chtít vrátit všechny nastavení na jejich výchozí hodnoty, stiskneme tlačítko *Výchozí* (10).



Obrázek A.3: Okno s nastavením aplikace s červeně vyznačenými elementy.

B Zdrojové kódy

Ze zdrojových kódů byly odebrány některé části, aby se alespoň mírně snížila jejich velikost. Byly odebrány direktivy překladače {$} a klauzule `uses`, které jen připojují ostatní jednotky. Dále byly odebrány prázdné metody ze zděděných tříd, volání rodičovských metod z potomků `inherited <metoda>` a popisy souborů (hlavičky). Kompletní zdrojové soubory jsou k dispozici na přiloženém CD.

Funkce, procedury a metody nejsou okomentovány v sekci `interfaces`, ale až v sekci `implementation`. Komentář se vždy vztahuje k následující řádce nebo řádkám a je psán zelenou kurzívou. Tmavě modré jsou klíčová slova Pascalu, světle modré jsou uživatelské metody, funkce a procedury a také přímé hodnoty. Zelené jsou ostatní systémové funkce a některé parametry metod.

B.1 UnitCommon.pas

```
unit UnitCommon;

interface

const
  // velikost hrací desky
  BoardSize = 8;
  // barvy kamenů obou hráčů
  StoneColors: array[1..2] of TColor = (clBlack, clWhite);
  // a jejich pojmenování
  StoneNames: array[1..2] of string = ('černý', 'bílý');
  // stavy políček na hrací desce
  btFree = 0;       // volné pole
  btStone1 = 1;     // pole s černým kamenem
  btStone2 = 2;     // pole s bílým kamenem
  // typy hráčů
  gtHuman = 0;              // člověk
  gtComputerEasy = 1;       // počítač (lehká obtížnost)
  gtComputerMedium = 2;     // počítač (střední obtížnost)
  gtComputerHard = 3;       // počítač (těžká obtížnost)
  gtNetwork = 4;            // síťový hráč
  // typ přerušení probíhající hry
  abNoAbort = 0;          // žádné přerušení
  abExit = 1;             // ukončení hry (ukončení aplikace, žádána nová
                          // hra...)
```

```
  abUndo = 2;             // uživatel vrací svůj předchozí tah
  abDisconnected = 3;     // došlo k přerušení síťové hry
  abEndGame = 4;          // nastal normální konec hry

type
  // další typy políček, které nesouvisí s vlastní hrou (grafické prvky
  // na hrací desce)
  // boPossibleStone1,2 - políčko je možným tahem pro daného hráče
  // boHighlight1,2 - na políčku se nachází kurzor myši (zvýraznění
  //                  políčka daného hráče)
  // boLastMove - na tomto políčku je naposledy položený kámen
  TBoardOverlays = (boPossibleStone1, boPossibleStone2, boHighlight1,
    boHighlight2, boLastMove);
  // a množina tvořená těmito prvky, neboť jich může být použito více
  // najednou
  TBoardOverlay = set of TBoardOverlays;
  // pole reprezentující jednotlivé stavy hry na hrací desce
  TBoardArray = array[0..BoardSize - 1, 0..BoardSize - 1] of byte;
  // pole reprezentující další stavy hry nesouvisející přímo se hrou
  // (grafické prvky) na hrací desce
  TBoardOverlayArray = array[0..BoardSize - 1, 0..BoardSize - 1] of
    TBoardOverlay;
  // tah hráče
  TMove = record
    x, y: byte;       // souřadnice tahu (x = sloupec, y = řádek)
    Player: byte;     // číslo hráče (1 - černý, 2 - bílý)
  end;
  // pole tahů (používané pro zjištění možných tahů)
  TMoveArray = array of TMove;
  // záznam v historii tahů
  THistory = record
    Board: TBoardArray;     // celý stav hrací desky před provedením tahu
    Move: TMove;            // tah
  end;
  // pole historie tahů
  THistoryArray = array of THistory;

function FirstUp(Text: string): string;
function CreateMove(x, y, Player: byte): TMove;
function AnotherPlayer(Player: integer): integer;

var
  // minimální čas (ms) než počítač zahraje tah (kvůli přehlednosti hry)
  CPUMinimumWait: integer = 1000;
  // hloubka prohledávání stromu algoritmu minimax
  MinimaxDepthMedium: integer = 2;  // u střední obtížnosti počítače
  MinimaxDepthHard: integer = 5;    // u těžké obtížnosti počítače
  // parametry ohodnocující funkce
  ValueField: integer = 1;        // ohodnocení zabraného políčka
```

```
  ValueEdge: integer = 100;        // políčko na hraně desky
  ValueCorner: integer = 10000;    // políčko v rohu desky
  ValueNoMove: integer = 100000;   // nemožnost provést tah
  ValueWin: integer = 1000000;     // výhra
  // algoritmus minimax může vrátit více tahů s nejlepším ohodnocením:
  // vybere pak z nich náhodně (true) nebo vezme první tah na seznamu
  // (false)
  RandomBestMoves: boolean = True;

implementation

// funkce vrátí text s prvním písmenem velkým (UTF-8 verze)
function FirstUp(Text: string): string;
var
  s: WideString;
begin
  s := UTF8Decode(Text); // je třeba UTF8 převést na UTF16...
  s := UpCase(s[1]) + Copy(s, 2, 100); // ...kde funkce UpCase funguje
  Result := UTF8Encode(s);
end;

// vytvoří a vrátí strukturu tah (ulehčení plnění struktury)
function CreateMove(x, y, Player: byte): TMove;
begin
  Result.x := x;
  Result.y := y;
  Result.Player := Player;
end;

// vrací číslo druhého hráče než který je v parametru Player
function AnotherPlayer(Player: integer): integer;
begin
  if Player = 1 then
    Result := 2
  else
    Result := 1;
end;

end.
```

B.2 UnitFormMain.pas

```
unit UnitFormMain;

interface

type
```

```
  // třída hlavního formuláře
  TFormMain = class(TForm)
    BGRAVirtualScreen: TBGRAVirtualScreen;
    MainMenu: TMainMenu;
    MenuItemUndoMove: TMenuItem;
    MenuItemTools: TMenuItem;
    MenuItemOptions: TMenuItem;
    MenuLineTools: TMenuItem;
    MenuLineGame: TMenuItem;
    MenuItemShowLastMove: TMenuItem;
    MenuItemView: TMenuItem;
    MenuItemShowPossibleMoves: TMenuItem;
    MenuItemNewGame: TMenuItem;
    MenuItemGame: TMenuItem;
    MenuItemQuit: TMenuItem;
    MenuItemHelp: TMenuItem;
    MenuItemAbout: TMenuItem;
    StatusBar: TStatusBar;
    procedure FormClose(Sender: TObject; var CloseAction: TCloseAction);
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure MenuItemOptionsClick(Sender: TObject);
    procedure MenuItemQuitClick(Sender: TObject);
    procedure MenuItemNewGameClick(Sender: TObject);
    procedure MenuItemAboutClick(Sender: TObject);
    procedure BGRAVirtualScreenRedraw(Sender: TObject;
      Bitmap: TBGRABitmap);
    procedure MenuItemShowLastMoveClick(Sender: TObject);
    procedure MenuItemShowPossibleMovesClick(Sender: TObject);
    procedure MenuItemUndoMoveClick(Sender: TObject);
    procedure StatusBarDrawPanel(AStatusBar: TStatusBar;
      Panel: TStatusPanel; const Rect: TRect);
    procedure StartNewGame;
  private
    // používá se, pokud chceme spustit novou a stávající stále běží
    NewGamePending: boolean;
  end;

var
  // instance formuláře
  FormMain: TFormMain;

implementation

// menu Hra/Konec - ukončení aplikace
procedure TFormMain.MenuItemQuitClick(Sender: TObject);
begin
  Close;
end;
```

```
// menu Hra/Nová hra - zobrazí formulář nové hry
procedure TFormMain.MenuItemNewGameClick(Sender: TObject);
begin
  if FormNewGame.ShowModal = mrOk then
  begin
    // pokud stávající hra běží
    if Assigned(Game) then begin
      // nastav přerušení hry
      Game.Abort := abExit;
      // a při ukončení hry se spustí nová
      NewGamePending := True;
    end
    else
      StartNewGame;
  end;
end;

// toto proběhne při vytvoření formuláře
procedure TFormMain.FormCreate(Sender: TObject);
begin
  DrawBoard := TDrawBoard.Create(BGRAVirtualScreen);
  NewGamePending := False;
end;

// toto proběhne při destrukci formuláře
procedure TFormMain.FormDestroy(Sender: TObject);
begin
  DrawBoard.Free;
  Game.Free;
end;

// menu Nástroje/Nastavení - zobrazí formulář nastavení
procedure TFormMain.MenuItemOptionsClick(Sender: TObject);
begin
  FormOptions.Show;
end;

// toto proběhne při zavření formuláře
procedure TFormMain.FormClose(Sender: TObject;
  var CloseAction: TCloseAction);
begin
  if Assigned(Game) then
    Game.Abort := abExit;
end;

// menu Nápověda/O programu - zobrazí informace o aplikaci
procedure TFormMain.MenuItemAboutClick(Sender: TObject);
begin
```

```
  FormAbout.Show;
end;

// toto je voláno při nutnosti překreslit plátno na formuláři
procedure TFormMain.BGRAVirtualScreenRedraw(Sender: TObject;
  Bitmap: TBGRABitmap);
begin
  DrawBoard.Draw;
end;

// menu Zobrazit/Poslední tah - zap/vyp zobrazení posledního tahu
procedure TFormMain.MenuItemShowLastMoveClick(Sender: TObject);
begin
  if Assigned(Game) then
    DrawBoard.RedrawBoard;
end;

// menu Zobrazit/Možné tahy - zap/vyp zobrazení možných tahů
procedure TFormMain.MenuItemShowPossibleMovesClick(Sender: TObject);
begin
  if Assigned(Game) then
    DrawBoard.RedrawBoard;
end;

// menu Hra/Tah zpět - provede zpět tah právě hrajícího hráče
procedure TFormMain.MenuItemUndoMoveClick(Sender: TObject);
begin
  if Assigned(Game) then
    Game.Abort := abUndo;
end;

// vykresluje panel ve stavovém řádku, dělám to manuálně, protože tam
// kreslím navíc kolečka s barvou hráče
procedure TFormMain.StatusBarDrawPanel(AStatusBar: TStatusBar;
  Panel: TStatusPanel; const Rect: TRect);
begin
  // WORKAROUND - i když je nastaveno, že budu sám vykreslovat stavový
  // řádek, text tam systém stejně vykreslil, je třeba tedy napřed celý
  // panel smazat
  AStatusBar.Canvas.Brush.Color := clDefault;
  AStatusBar.Canvas.FillRect(Rect);
  // kreslíme pouze pokud je v panelu text
  if Panel.Text <> '' then begin
    // vykreslení kolečka
    AStatusBar.Canvas.Pen.Color := clBlack;
    if Panel = AStatusBar.Panels[0] then
      AStatusBar.Canvas.Brush.Color := StoneColors[1]
    else
      AStatusBar.Canvas.Brush.Color := StoneColors[2];
```

```
    AStatusBar.Canvas.Ellipse(Rect.Left + 2, Rect.Top +
      2, Rect.Left + Rect.Bottom - Rect.Top - 2, Rect.Bottom - 2);
    // vykreslení textu
    AStatusBar.Canvas.Brush.Color := clDefault;
    AStatusBar.Canvas.Font.Bold := True;
    AStatusBar.Canvas.TextOut(Rect.Left + Rect.Bottom -
      Rect.Top + 3, 3, Panel.Text);
  end;
end;

// spuštění nové hry
procedure TFormMain.StartNewGame;
var
  i: integer;
  BackToNewGame: boolean = False; // indikuje návrat z nastavení sítě
begin
  // vytvoření instance hry
  Game := TGame.Create;
  DrawBoard.RedrawBoard;
  // vytvoření instancí hráčů na základě nastavení z formuláře Nová hra
  for i := 1 to 2 do begin
    case FormNewGame.Choice[i] of
      gtHuman:
        Players[i] := TPlayerHuman.Create(i);
      gtComputerEasy:
        Players[i] := TPlayerCPURandom.Create(i);
      gtComputerMedium:
        Players[i] := TPlayerCPUMinimax.Create(i, MinimaxDepthMedium);
      gtComputerHard:
        Players[i] := TPlayerCPUMinimax.Create(i, MinimaxDepthHard);
      gtNetwork:
        Players[i] := TPlayerNetwork.Create(i);
    end;
    Players[i].Init;
  end;
  if Game.Abort = abNoAbort then
  begin
    // tah zpět nelze dělat při síťové hře
    MenuItemUndoMove.Enabled :=
      (FormNewGame.Choice[1] <> gtNetwork) and
      (FormNewGame.Choice[2] <> gtNetwork);
    Game.Play;
    MenuItemUndoMove.Enabled := False;
  end
  else
    BackToNewGame := True;
  FreeAndNil(Game);
  for i := 1 to 2 do
    Players[i].Free;
```

```
  // pokud je očekávána nová hra, nyní po zrušení všech instancí hry a
  // hráčů ji můžeme opět spustit
  if NewGamePending then begin
    NewGamePending := False;
    StartNewGame;
  end;
  // pokud byl zrušen dialog sítě, zobraz opět dialog nové hry
  if BackToNewGame then
    MenuItemNewGameClick(nil);
end;

end.
```

B.3 UnitFormOptions.pas

```
unit UnitFormOptions;

interface

type
  // třída formuláře
  TFormOptions = class(TForm)
    ButtonDefaults: TButton;
    ButtonOK: TButton;
    ButtonCancel: TButton;
    CheckBoxRandomBestMoves: TCheckBox;
    LabeledEditCPUMinimumWait: TLabeledEdit;
    LabeledEditMediumDepth: TLabeledEdit;
    LabeledEditHardDepth: TLabeledEdit;
    LabeledEditValueField: TLabeledEdit;
    LabeledEditValueEdge: TLabeledEdit;
    LabeledEditValueCorner: TLabeledEdit;
    LabeledEditValueNoMove: TLabeledEdit;
    LabeledEditValueWin: TLabeledEdit;
    procedure ButtonCancelClick(Sender: TObject);
    procedure ButtonDefaultsClick(Sender: TObject);
    procedure ButtonOKClick(Sender: TObject);
    procedure FormShow(Sender: TObject);
  end;

var
  // instance formuláře
  FormOptions: TFormOptions;

implementation

// toto proběhne při zobrazení formuláře - jednotlivá editovatelná pole
```

```
// se vyplní z globálních proměnných
procedure TFormOptions.FormShow(Sender: TObject);
begin
  LabeledEditCPUMinimumWait.Text := IntToStr(CPUMinimumWait);
  LabeledEditMediumDepth.Text := IntToStr(MinimaxDepthMedium);
  LabeledEditHardDepth.Text := IntToStr(MinimaxDepthHard);
  LabeledEditValueField.Text := IntToStr(ValueField);
  LabeledEditValueEdge.Text := IntToStr(ValueEdge);
  LabeledEditValueCorner.Text := IntToStr(ValueCorner);
  LabeledEditValueNoMove.Text := IntToStr(ValueNoMove);
  LabeledEditValueWin.Text := IntToStr(ValueWin);
  CheckBoxRandomBestMoves.Checked := RandomBestMoves;
  LabeledEditCPUMinimumWait.SetFocus;
end;

// kliknutí na tlačítko OK - všechny hodnoty se uloží do globálních
// proměnných, pokud je nějaká hodnota špatně zadaná, kurzor skočí na
// ni a formulář se nezavře
procedure TFormOptions.ButtonOKClick(Sender: TObject);
var
  i: integer;
begin
  i := StrToIntDef(LabeledEditCPUMinimumWait.Text, -1);
  if i > 0 then
    CPUMinimumWait := i
  else begin
    LabeledEditCPUMinimumWait.SetFocus;
    Exit;
  end;
  i := StrToIntDef(LabeledEditMediumDepth.Text, -1);
  if i > 0 then
    MinimaxDepthMedium := i
  else begin
    LabeledEditMediumDepth.SetFocus;
    Exit;
  end;
  i := StrToIntDef(LabeledEditHardDepth.Text, -1);
  if i > 0 then
    MinimaxDepthHard := i
  else begin
    LabeledEditHardDepth.SetFocus;
    Exit;
  end;
  i := StrToIntDef(LabeledEditValueField.Text, -1);
  if i >= 0 then
    ValueField := i
  else begin
    LabeledEditValueField.SetFocus;
    Exit;
```

```
  end;
  i := StrToIntDef(LabeledEditValueEdge.Text, -1);
  if i >= 0 then
    ValueEdge := i
  else begin
    LabeledEditValueEdge.SetFocus;
    Exit;
  end;
  i := StrToIntDef(LabeledEditValueCorner.Text, -1);
  if i >= 0 then
    ValueCorner := i
  else begin
    LabeledEditValueCorner.SetFocus;
    Exit;
  end;
  i := StrToIntDef(LabeledEditValueNoMove.Text, -1);
  if i >= 0 then
    ValueNoMove := i
  else begin
    LabeledEditValueNoMove.SetFocus;
    Exit;
  end;
  i := StrToIntDef(LabeledEditValueNoMove.Text, -1);
  if i >= 0 then
    ValueNoMove := i
  else begin
    LabeledEditValueNoMove.SetFocus;
    Exit;
  end;
  i := StrToIntDef(LabeledEditValueWin.Text, -1);
  if i >= 0 then
    ValueWin := i
  else begin
    LabeledEditValueWin.SetFocus;
    Exit;
  end;
  RandomBestMoves := CheckBoxRandomBestMoves.Checked;
  Close;
end;

// kliknutí na tlačítko Storno - zavření formuláře bez uložení hodnot
procedure TFormOptions.ButtonCancelClick(Sender: TObject);
begin
  Close;
end;

// kliknutí na tlačítko Výchozí - nastavení výchozích hodnot
procedure TFormOptions.ButtonDefaultsClick(Sender: TObject);
begin
```

```
  LabeledEditCPUMinimumWait.Text := '1000';
  LabeledEditMediumDepth.Text := '2';
  LabeledEditHardDepth.Text := '5';
  LabeledEditValueField.Text := '1';
  LabeledEditValueEdge.Text := '100';
  LabeledEditValueCorner.Text := '10000';
  LabeledEditValueNoMove.Text := '100000';
  LabeledEditValueWin.Text := '1000000';
  CheckBoxRandomBestMoves.Checked := True;
end;

end.
```

B.4 UnitFormNewGame.pas

```
unit UnitFormNewGame;

interface

type
  // třída formuláře
  TFormNewGame = class(TForm)
    GroupBox2: TGroupBox;
    RB24: TRadioButton;
    RB20: TRadioButton;
    RB21: TRadioButton;
    RB22: TRadioButton;
    RB23: TRadioButton;
    SpeedButtonChange: TSpeedButton;
    VirtualScreen1: TBGRAVirtualScreen;
    ButtonNewGame: TButton;
    ButtonCancel: TButton;
    GroupBox1: TGroupBox;
    RB10: TRadioButton;
    RB11: TRadioButton;
    RB12: TRadioButton;
    RB13: TRadioButton;
    RB14: TRadioButton;
    VirtualScreen2: TBGRAVirtualScreen;
    procedure RBClick(Sender: TObject);
    procedure SpeedButtonChangeClick(Sender: TObject);
    procedure VirtualScreen1Redraw(Sender: TObject;
      Bitmap: TBGRABitmap);
    procedure VirtualScreen2Redraw(Sender: TObject;
      Bitmap: TBGRABitmap);
  public
    // bude obsahovat typ hráče 1 a typ hráče 2 po zavření formuláře
```

```
    Choice: array[1..2] of byte;
  end;

var
  // instance formuláře
  FormNewGame: TFormNewGame;

implementation

// vykresluje bílé kolečko
procedure TFormNewGame.VirtualScreen2Redraw(Sender: TObject;
  Bitmap: TBGRABitmap);
begin
  VirtualScreen2.Bitmap.FillEllipseAntialias(24, 24, 22, 22,
    ColorToBGRA(StoneColors[2]));
  VirtualScreen2.Bitmap.EllipseAntialias(24, 24, 22, 22,
    ColorToBGRA(clBlack), 2);
end;

// vykresluje černé kolečko
procedure TFormNewGame.VirtualScreen1Redraw(Sender: TObject;
  Bitmap: TBGRABitmap);
begin
  VirtualScreen1.Bitmap.FillEllipseAntialias(24, 24, 22, 22,
    ColorToBGRA(StoneColors[1]));
  VirtualScreen1.Bitmap.EllipseAntialias(24, 24, 22, 22,
    ColorToBGRA(clBlack), 2);
end;

// kliknutí na jakýkoliv přepínač typu hráče
procedure TFormNewGame.RBClick(Sender: TObject);
begin
  // z názvu komponenty RBxy zjistí číslo hráče x a číslo typu hráče y
  Choice[StrToInt(TRadioButton(Sender).Name[3])] :=
    StrToInt(TRadioButton(Sender).Name[4]);
  // pokud je vybrán síťový hráč, znemožni výběr síťového hráče i pro
  // druhého hráče
  if RB14.Checked then
    RB24.Enabled := False
  else
    RB24.Enabled := True;
  if RB24.Checked then
    RB14.Enabled := False
  else
    RB14.Enabled := True;
end;

// kliknutí na tlačítko "<>" - prohození typů hráče
procedure TFormNewGame.SpeedButtonChangeClick(Sender: TObject);
```

```
var
  i: integer;
  rb1, rb2: TRadioButton;
  frb1, frb2: TRadioButton;
begin
  // projede všechny přepínače a zjistí, které 2 jsou sepnuty; zapamatuje
  // si jejich protějšek
  for i := 0 to 4 do begin
    rb1 := TRadioButton(FindComponent('RB1' + IntToStr(i)));
    rb2 := TRadioButton(FindComponent('RB2' + IntToStr(i)));
    if rb1.Checked then
      frb2 := rb2;
    if rb2.Checked then
      frb1 := rb1;
  end;
  // sepne pak jejich protějšky
  frb1.Checked := True;
  frb2.Checked := True;
end;

end.
```

B.5 UnitFormEndGame.pas

```
unit UnitFormEndGame;

interface

type
  // třída formuláře
  TFormEndGame = class(TForm)
    VirtualScreen: TBGRAVirtualScreen;
    ButtonRetry: TButton;
    ButtonNew: TButton;
    ButtonClose: TButton;
    LabelText: TLabel;
    procedure VirtualScreenRedraw(Sender: TObject; Bitmap: TBGRABitmap);
    procedure ButtonCloseClick(Sender: TObject);
    procedure ButtonNewClick(Sender: TObject);
    procedure ButtonRetryClick(Sender: TObject);
  private
    FWinner: integer;
  public
    property Winner: integer read FWinner write FWinner;
  end;

var
```

```
  // instance formuláře
  FormEndGame: TFormEndGame;

implementation

// zobrazí barevné kolečko na základě výsledku hry
procedure TFormEndGame.VirtualScreenRedraw(Sender: TObject;
  Bitmap: TBGRABitmap);
begin
  // pokud byla hra ukončena dříve, nic nezobraz
  if Winner = -1 then
    Exit;
  if Winner > 0 then
    // kolečko s barvou vítěze
    VirtualScreen.Bitmap.FillEllipseAntialias(29, 29, 25,
      25, ColorToBGRA(StoneColors[Winner]))
  else begin
    // šedé kolečko - remíza
    VirtualScreen.Bitmap.FillEllipseAntialias(29, 29, 25,
      25, ColorToBGRA(clGray));
  end;
  // černý lem kolečka
  VirtualScreen.Bitmap.EllipseAntialias(29, 29, 25, 25,
    ColorToBGRA(clBlack), 2);
end;

// kliknutí na tlačítko Zavřít - zavření formuláře
procedure TFormEndGame.ButtonCloseClick(Sender: TObject);
begin
  Close;
end;

// kliknutí na tlačítko "Nová hra" - zobrazení formuláře nové hry
procedure TFormEndGame.ButtonNewClick(Sender: TObject);
begin
  Close;
  FormMain.MenuItemNewGameClick(nil);
end;

// kliknutí na tlačítko "Opakovat hru" - nová hra se stejnými typy
// hráčů jako právě proběhlá hra
procedure TFormEndGame.ButtonRetryClick(Sender: TObject);
begin
  Close;
  FormMain.StartNewGame;
end;

end.
```

B.6 UnitFormAbout.pas

```
unit UnitFormAbout;

interface

type
  // třída formuláře
  TFormAbout = class(TForm)
    ButtonOK: TButton;
    Image: TImage;
    LabelOthello: TLabel;
    LabelName: TLabel;
    LabelBy: TLabel;
    procedure ButtonOKClick(Sender: TObject);
  end;

var
  // instance formuláře
  FormAbout: TFormAbout;

implementation

// kliknutí na tlačítko OK - zavře formulář
procedure TFormAbout.ButtonOKClick(Sender: TObject);
begin
  Close;
end;

end.
```

B.7 UnitDrawBoard.pas

```
unit UnitDrawBoard;

interface

type
  // třída
  TDrawBoard = class
  private
    // vypočtené délky okrajů (zleva, shora)
    BorderX, BorderY: integer;
    // vypočtená délka políčka
    FieldLength: integer;
    // plátno na které kreslíme
```

```
    FVirtualScreen: TBGRAVirtualScreen;
  const
    // minimální délka okrajů
    MinimumBorder = 10;
  public
    constructor Create(AVirtualScreen: TBGRAVirtualScreen);
    procedure Draw;
    function GetScreenX(FieldX: integer): integer;
    function GetScreenY(FieldY: integer): integer;
    function GetBoardX(x: integer): integer;
    function GetBoardY(y: integer): integer;
    procedure DrawField(FieldX, FieldY: integer; RedrawScreen: boolean = True);
    procedure RedrawBoard;
    function GetScreenRect(FieldX, FieldY: integer): TRect;
    procedure CaptureMouse(MoveProcedure: TMouseMoveEvent;
      ClickProcedure: TNotifyEvent);
    procedure ReleaseMouse;
  end;

var
  // instance
  DrawBoard: TDrawBoard;

implementation

// vrací grafickou souřadnici x políčka ve sloupci FieldX
function TDrawBoard.GetScreenX(FieldX: integer): integer;
begin
  Result := BorderX + FieldX * FieldLength;
end;

// vrací grafickou souřadnici y políčka v řádce FieldY
function TDrawBoard.GetScreenY(FieldY: integer): integer;
begin
  Result := BorderY + FieldY * FieldLength;
end;

// vrací grafické souřadnice obdélníku, který reprezentuje políčko na
// souřadnicích [FieldX, FieldY]
function TDrawBoard.GetScreenRect(FieldX, FieldY: integer): TRect;
begin
  Result := Bounds(GetScreenX(FieldX), GetScreenY(FieldY), FieldLength -
    1, FieldLength - 1);
end;

// vrací sloupec políčka, které se nachazí pod grafickou souřadnicí x
function TDrawBoard.GetBoardX(x: integer): integer;
begin
  Result := (x - BorderX) div FieldLength;
```

```
end;

// vrací řádek políčka, které se nachazí pod grafickou souřadnicí y
function TDrawBoard.GetBoardY(y: integer): integer;
begin
  Result := (y - BorderY) div FieldLength;
end;

// pohyb a klikání myši po formuláři bude vyhodnocován v těchto metodách
procedure TDrawBoard.CaptureMouse(MoveProcedure: TMouseMoveEvent;
  ClickProcedure: TNotifyEvent);
begin
  FVirtualScreen.OnMouseMove := MoveProcedure;
  FVirtualScreen.OnClick := ClickProcedure;
end;

// pohyb a klikání myši po formuláři nebude vyhodnocováno
procedure TDrawBoard.ReleaseMouse;
begin
  FVirtualScreen.OnMouseMove := nil;
  FVirtualScreen.OnClick := nil;
end;

// vykreslení políčka [FieldX = sloupec, FieldY = řádek] hrací plochy
// RedrawScreen - pokud True, pak po vykreslení ihned plátno na formulář
procedure TDrawBoard.DrawField(FieldX, FieldY: integer;
  RedrawScreen: boolean = True);
begin
  // vykresli zelené prázdné políčko
  FVirtualScreen.Bitmap.FillRect(GetScreenRect(FieldX, FieldY), clGreen);
  // dál pokračuj pouze pokud se právě hraje hra
  if Assigned(Game) then begin
    case Game.Board.Board[FieldX, FieldY] of
      // vykresli kámen
      btStone1, btStone2: begin
        FVirtualScreen.Bitmap.FillEllipseAntialias(
          GetScreenX(FieldX) + FieldLength div 2, GetScreenY(FieldY) +
          FieldLength div 2, Round(FieldLength / 2.5),
          Round(FieldLength / 2.5),
          ColorToBGRA(StoneColors[Game.Board.Board[FieldX, FieldY]]));
        FVirtualScreen.Bitmap.FillEllipseAntialias(
          GetScreenX(FieldX) + FieldLength div 2, GetScreenY(FieldY) +
          FieldLength div 2, Round(FieldLength / 3), Round(FieldLength / 3),
          BGRA(128, 128, 128, 16));
      end;
    end;
    // pokud je zaplé zobrazení možných tahů, vykresli je
    if FormMain.MenuItemShowPossibleMoves.Checked then begin
      if boPossibleStone1 in Game.Overlay[FieldX, FieldY] then
```

```
        FVirtualScreen.Bitmap.FillRect(GetScreenRect(FieldX, FieldY),
          ColorToBGRA(StoneColors[1], 96), dmLinearBlend);
      if boPossibleStone2 in Game.Overlay[FieldX, FieldY] then
        FVirtualScreen.Bitmap.FillRect(GetScreenRect(FieldX, FieldY),
          ColorToBGRA(StoneColors[2], 96), dmLinearBlend);
    end;
    // pokud je zaplé zobrazení posledního tahu, vykresli ho
    if FormMain.MenuItemShowLastMove.Checked and
      (boLastMove in Game.Overlay[FieldX, FieldY]) then
      FVirtualScreen.Bitmap.EllipseAntialias(GetScreenX(FieldX) +
        FieldLength div 2, GetScreenY(FieldY) + FieldLength div 2,
        Round(FieldLength / 2.75), Round(FieldLength / 2.75),
        BGRA(0, 255, 0), (FieldLength / 2.4 - FieldLength / 3));
    // zvýraznění políčka, na které právě ukazuje myš
    if boHighlight1 in Game.Overlay[FieldX, FieldY] then
      FVirtualScreen.Bitmap.FillRect(GetScreenRect(FieldX, FieldY),
        ColorToBGRA(StoneColors[1], 64), dmLinearBlend);
    if boHighlight2 in Game.Overlay[FieldX, FieldY] then
      FVirtualScreen.Bitmap.FillRect(GetScreenRect(FieldX, FieldY),
        ColorToBGRA(StoneColors[2], 64), dmLinearBlend);
  end;
  if RedrawScreen then
    FVirtualScreen.Invalidate;
end;

// překresli hrací desku na plátno
procedure TDrawBoard.RedrawBoard;
begin
  FVirtualScreen.RedrawBitmap;
  Application.ProcessMessages;
end;

// vykreslí všechna políčka hrací desky na plátno a vypočte hodnoty
// okrajů a velikosti políčka na základě velikosti plátna (formuláře)
procedure TDrawBoard.Draw;
var
  x, y: integer;
begin
  if FVirtualScreen.Width > FVirtualScreen.Height then
    FieldLength := FVirtualScreen.Height - 2 * MinimumBorder
  else
    FieldLength := FVirtualScreen.Width - 2 * MinimumBorder;
  BorderX := (FVirtualScreen.Width - FieldLength) div 2;
  BorderY := (FVirtualScreen.Height - FieldLength) div 2;
  FieldLength := FieldLength div BoardSize;
  for y := 0 to BoardSize - 1 do
    for x := 0 to BoardSize - 1 do
      DrawField(x, y, False);
end;
```

```
// konstruktor - jako parametr je plátno, na které se bude kreslit
constructor TDrawBoard.Create(AVirtualScreen: TBGRAVirtualScreen);
begin
  FVirtualScreen := AVirtualScreen;
end;

end.
```

B.8 UnitBoard.pas

```
unit UnitBoard;

interface

type
  // třída
  TBoard = class
  private
    // pole hrací plochy
    FBoard: TBoardArray;
    function CheckPossibleMove(Move: TMove): boolean;
  public
    procedure DoMove(Move: TMove);
    function FindPossibleMoves(MovingPlayer: byte): TMoveArray;
    constructor Create;
    // přímý přístup k poli
    property Board: TBoardArray read FBoard write FBoard;
  end;

implementation

// provede tah, tj. umístí kámen a zajme všechny možné soupeřovo kameny
procedure TBoard.DoMove(Move: TMove);
var
  i, j: integer;
  x, y: integer;
begin
  if FBoard[Move.x, Move.y] > btFree then
    Exit;
  // umístí kámen
  FBoard[Move.x, Move.y] := Move.Player;
  // projede všechny směry [-1..1,-1..1]
  for j := -1 to 1 do
    for i := -1 to 1 do begin
      // kromě směru [0,0]
      if (i = 0) and (j = 0) then
```

```
        Continue;
      x := Move.x + i;
      y := Move.y + j;
      // dokud nenarazí na okraj plochy
      while (x >= 0) and (x < BoardSize) and (y >= 0) and
        (y < BoardSize) do begin
        // nebo prázdné pole
        if FBoard[x, y] = btFree then
          Break;
        // nebo na vlastní kámen
        if FBoard[x, y] = Move.Player then begin
          x := x - i;
          y := y - j;
          // rozjede se zase zpět a všechny soupeřovo kameny přebarví
          // na vlastní
          while (x <> Move.x) or (y <> Move.y) do begin
            FBoard[x, y] := Move.Player;
            x := x - i;
            y := y - j;
          end;
          Break;
        end;
        // sem se dostane pouze pokud narazil na soupeřovo kámen
        x := x + i;
        y := y + j;
      end;
    end;
end;

// najde všechny možné tahy hráče číslo MovingPlayer a vrátí je v poli
function TBoard.FindPossibleMoves(MovingPlayer: byte): TMoveArray;
var
  x, y: integer;
  // počet nalezených tahů
  Count: integer = 0;
begin
  // může být maximálně 60 tahů
  SetLength(Result, 60);
  // projede políčko po políčku a otestuje, zda-li tam může být proveden
  // tah
  for y := 0 to BoardSize - 1 do
    for x := 0 to BoardSize - 1 do begin
      if CheckPossibleMove(CreateMove(x, y, MovingPlayer)) then begin
        Inc(Count);
        Result[Count - 1] := CreateMove(x, y, MovingPlayer);
      end;
    end;
  // velikost pole upravíme na skutečný počet pruků
  SetLength(Result, Count);
```

```
end;

// zkoumá, zda-li je možné provést tah Move
function TBoard.CheckPossibleMove(Move: TMove): boolean;
var
  i, j: integer;
  x, y: integer;
  AnotherStone: boolean;
begin
  Result := False;
  // pokud je již zde nějaký kámen položen, nemá cenu dál zkoumat
  if FBoard[Move.x, Move.y] > btFree then
    Exit;
  // projede všechny směry [-1..1,-1..1] od zamýšleného tahu
  for j := -1 to 1 do
    for i := -1 to 1 do begin
      // kromě směru [0,0]
      if (i = 0) and (j = 0) then
        Continue;
      x := Move.x + i;
      y := Move.y + j;
      AnotherStone := False;
      // dokud nenarazí na okraj desky
      while (x >= 0) and (x < BoardSize) and (y >= 0) and
        (y < BoardSize) do begin
        // nebo volné pole
        if FBoard[x, y] = btFree then
          Break;
        // nebo vlastní kámen
        if FBoard[x, y] = Move.Player then
          // pokud již před tím narazil na soupeřovo kámen, tah je OK
          if AnotherStone then begin
            Result := True;
            Exit;
          end
          else
            Break;
        // narazil na soupeřovo kámen
        x := x + i;
        y := y + j;
        AnotherStone := True;
      end;
    end;
end;

// konstruktor - nastaví všechna políčka jako volná
constructor TBoard.Create;
var
  x, y: integer;
```

```
begin
  for y := 0 to BoardSize - 1 do
    for x := 0 to BoardSize - 1 do
      FBoard[x, y] := btFree;
end;

end.
```

B.9 UnitGame.pas

```
unit UnitGame;

interface

type
  // třída
  TGame = class
  private
    FAbort: byte;
    FOverlay: TBoardOverlayArray;
    PossibleMoves: TMoveArray;
    MovingPlayer: integer;
    FHistory: THistoryArray;
    FBoard: TBoard;
    procedure DoUndo;
    procedure SetAbort(const AValue: byte);
    procedure ShowEndDialog;
    procedure ShowStatus;
    procedure SetPossibleMoves;
    function CheckMove(Move: TMove): boolean;
  public
    constructor Create;
    procedure Play;
    property Abort: byte read FAbort write SetAbort;
    property Board: TBoard read FBoard;
    property Overlay: TBoardOverlayArray read FOverlay;
    property History: THistoryArray read FHistory;
  end;

var
  // instance
  Game: TGame;

implementation

// konstruktor - založí pole hrací desky a pole dalších typů políček
constructor TGame.Create;
```

```
var
  i, j: integer;
begin
  FAbort := abNoAbort;
  FBoard := TBoard.Create;
  SetLength(FHistory, 0);
  for j := 0 to BoardSize - 1 do
    for i := 0 to BoardSize - 1 do
      FOverlay[i, j] := [];
  for i := 0 to 2 do
    FormMain.StatusBar.Panels[i].Text := '';
end;

// nastaví přerušení hry typu AValue
procedure TGame.SetAbort(const AValue: byte);
begin
  if FAbort = AValue then
    Exit;
  FAbort := AValue;
end;

// provede vrácení minulého tahu aktuálního hráče
procedure TGame.DoUndo;
var
  // index do pole historie
  Idx: integer;
  // nalezen minulý tah tohoto hráče?
  Found: boolean = False;
begin
  Idx := Length(FHistory);
  // projede pole historie a hledá, kde je poslední hráčův tah
  while Idx > 0 do begin
    if FHistory[Idx - 1].Move.Player = MovingPlayer then begin
      Found := True;
      Break;
    end;
    Dec(Idx);
  end;
  // pokud ho nalezl, pak vrátí stav hracího pole do tohoto okamžiku
  if Found then begin
    // indikace posledního tahu je zrušena
    Exclude(FOverlay[FHistory[Length(FHistory) - 1].Move.x,
      FHistory[Length(FHistory) - 1].Move.y], boLastMove);
    FBoard.Board := FHistory[Idx - 1].Board;
    // pokud nejsme ve výchozím stavu, pak nastav indikaci posledního
    // tahu
    if Idx > 1 then
      Include(FOverlay[FHistory[Idx - 2].Move.x,
        FHistory[Idx - 2].Move.y], boLastMove);
```

```
    // pole historie zkraťˇ na současnou délku
    SetLength(FHistory, Idx - 1);
  end;
end;

// smyčka hry
procedure TGame.Play;
var
  PlayerMove: TMove;
begin
  // nastavení výchozího umístění kamenů
  FBoard.Board[BoardSize div 2 - 1, BoardSize div 2 - 1] := btStone1;
  FBoard.Board[BoardSize div 2, BoardSize div 2 - 1] := btStone2;
  FBoard.Board[BoardSize div 2 - 1, BoardSize div 2] := btStone2;
  FBoard.Board[BoardSize div 2, BoardSize div 2] := btStone1;
  MovingPlayer := 1;
  Players[2].Pass := False;
  repeat
    ShowStatus;
    Players[MovingPlayer].Pass := False;
    // nalezne možné tahy hráče, který je na tahu
    PossibleMoves := FBoard.FindPossibleMoves(MovingPlayer);
    SetPossibleMoves;
    DrawBoard.RedrawBoard;
    // provedení začátku hraní hráče (jako parametr je indikace
    // nemožnosti hrát)
    Players[MovingPlayer].MoveBegin(Length(PossibleMoves) = 0);
    // pokud jsou k dispozici legální tahy
    if Length(PossibleMoves) > 0 then begin
      // budeme čekat, dokud hráč neodehraje legální tah nebo nenastane
      // přerušení hry
      repeat
        PlayerMove := Players[MovingPlayer].Moving;
      until (Abort > abNoAbort) or CheckMove(PlayerMove);
      // provedení konce tahu hráče
      Players[MovingPlayer].MoveEnd;
      // pokud nebylo přerušeno
      if Abort = abNoAbort then begin
        // přidáme tah do historie
        SetLength(FHistory, Length(FHistory) + 1);
        FHistory[Length(FHistory) - 1].Move := PlayerMove;
        FHistory[Length(FHistory) - 1].Board := FBoard.Board;
        // zrušíme nastavení typu poslední tah na minulém políčku
        if Length(FHistory) > 1 then
          Exclude(FOverlay[FHistory[Length(FHistory) - 2].Move.x,
            FHistory[Length(FHistory) - 2].Move.y], boLastMove);
        // provede se tah
        FBoard.DoMove(PlayerMove);
        // nastavíme typ poslední tah na políčko
```

```
        Include(FOverlay[PlayerMove.x, PlayerMove.y], boLastMove);
      end;
    end;
    // pokud nastalo přerušení kvůli vrácení posledního tahu
    if Abort = abUndo then begin
      Abort := abNoAbort;
      // proveď vrácení tahu
      DoUndo;
      MovingPlayer := AnotherPlayer(MovingPlayer);
    end;
    // na tahu je další hráč
    if Abort = abNoAbort then
      MovingPlayer := AnotherPlayer(MovingPlayer);
    // pokud ani jeden hráč nemohl hrát, pak nastává přerušení typu
    // konec hry
    if Players[1].Pass and Players[2].Pass then
      Abort := abEndGame;
  until Abort > abNoAbort;
  Players[1].Deinit;
  Players[2].Deinit;
  // pokud hra skončila ukončením aplikace nebo začátkem nové hry, pak
  // nezobrazuj dialog konce hry
  if Abort = abExit then
    Exit;
  DrawBoard.RedrawBoard;
  ShowStatus;
  ShowEndDialog;
end;

// zobrazí dialog konce hry
procedure TGame.ShowEndDialog;
var
  i: integer;
begin
  with FormEndGame do begin
    // nastala remíza
    if Players[1].Score = Players[2].Score then begin
      LabelText.Caption :=
        Format('Remíza %d:%d', [Players[1].Score, Players[2].Score]);
      Winner := 0;
    end
    else begin
      // vyhrál hráč 1 nebo 2
      if Players[1].Score > Players[2].Score then
        i := 1
      else
        i := 2;
      LabelText.Caption :=
        Format('%s vyhrál %d:%d', [FirstUp(StoneNames[i]), BoardSize *
```

```
        BoardSize - Players[AnotherPlayer(i)].Score,
        Players[AnotherPlayer(i)].Score]);
      Winner := i;
    end;
    // před koncem hry nastalo přerušení spojení, to je třeba dát vědět
    if Abort = abDisconnected then begin
      LabelText.Caption := 'Spojení přerušeno!';
      Winner := -1;
    end;
    Show;
  end;
end;

// vypočítá současné skóre hráčů a vypíše stav hry do stavového řádku
// formuláře
procedure TGame.ShowStatus;
var
  x, y: integer;
begin
  Players[1].Score := 0;
  Players[2].Score := 0;
  // sečte skóre (kameny) jednotlivých hráčů
  for y := 0 to BoardSize - 1 do
    for x := 0 to BoardSize - 1 do
      if FBoard.Board[x, y] > btFree then
        Players[FBoard.Board[x, y]].Score :=
          Players[FBoard.Board[x, y]].Score + 1;
  for x := 1 to 2 do
    FormMain.StatusBar.Panels[x - 1].Text := IntToStr(Players[x].Score);
  // zobrazí stav hry
  // konec hry
  if Abort = abEndGame then
    FormMain.StatusBar.Panels[2].Text := 'Konec hry'
  else
  // hráč nemohl tahnout, protože neměl žádný legální tah
  if Players[AnotherPlayer(MovingPlayer)].Pass then
    FormMain.StatusBar.Panels[2].Text :=
      Format('%s nemohl táhnout, na tahu je opět %s',
      [FirstUp(StoneNames[AnotherPlayer(MovingPlayer)]),
      StoneNames[MovingPlayer]])
  else
    // nebo vypíše kdo je na tahu
    FormMain.StatusBar.Panels[2].Text :=
      Format('Na tahu je %s', [StoneNames[MovingPlayer]]);
  Application.ProcessMessages;
end;

// nastaví typ "možné tahy" do pole dalších typů
procedure TGame.SetPossibleMoves;
```

```
var
  i, j: integer;
begin
  // napřed všechny "možné tahy" smaže
  for j := 0 to BoardSize - 1 do
    for i := 0 to BoardSize - 1 do
      FOverlay[i, j] := FOverlay[i, j] -
        [boPossibleStone1, boPossibleStone2];
  // pak je opět označí z pole možných tahů
  for i := 0 to Length(PossibleMoves) - 1 do
    if MovingPlayer = 1 then
      Include(FOverlay[PossibleMoves[i].x, PossibleMoves[i].y],
        boPossibleStone1)
    else
      Include(FOverlay[PossibleMoves[i].x, PossibleMoves[i].y],
        boPossibleStone2);
end;

// testuje, zda-li daný tah Move je legální
function TGame.CheckMove(Move: TMove): boolean;
var
  i: integer;
begin
  Result := False;
  // porovnává, zda-li je tah Move v poli možných tahů
  for i := 0 to Length(PossibleMoves) - 1 do
    if (Move.x = PossibleMoves[i].x) and
      (Move.y = PossibleMoves[i].y) then begin
      Result := True;
      Exit;
    end;
end;

end.
```

B.10 UnitPlayer.pas

```
unit UnitPlayer;

interface

type
  // třída
  TPlayer = class
  private
    FPass: boolean;
    FScore: integer;
```

```
  protected
    FNumber: integer;
  public
    constructor Create(PlayerNumber: integer); virtual;
    procedure MoveBegin(IsPass: boolean); virtual;
    // abstraktní metody budou definovány až v potomcích třídy
    function Moving: TMove; virtual; abstract;
    procedure MoveEnd; virtual; abstract;
    procedure Init; virtual; abstract;
    procedure Deinit; virtual; abstract;
    // číslo hráče
    property Number: integer read FNumber;
    // indikace, zda-li hráč nemůže hrát
    property Pass: boolean read FPass write FPass;
    // současné skóre hráče
    property Score: integer read FScore write FScore;
  end;

var
  // dvě instance (dva hráči)
  Players: array[1..2] of TPlayer;

implementation

// konstruktor - nastavení výchozích hodnot
constructor TPlayer.Create(PlayerNumber: integer);
begin
  FNumber := PlayerNumber;
  FPass := False;
  FScore := 0;
end;

// začátek tahu - nastaví, jestli hráč mohl hrát
procedure TPlayer.MoveBegin(IsPass: boolean);
begin
  FPass := IsPass;
end;

end.
```

B.11 UnitPlayerHuman.pas

```
unit UnitPlayerHuman;

interface

type
```

```
  // třída
  TPlayerHuman = class(TPlayer)
  private
    // předchozí pozice políčka hrací desky, kde byl kurzor myši
    OldMouseX, OldMouseY: integer;
    // uživatel stiskl tlačítko myši
    Clicked: boolean;
    procedure MouseClick(Sender: TObject);
    procedure MouseMove(Sender: TObject; Shift: TShiftState;
      x, y: integer);
  public
    constructor Create(PlayerNumber: integer); override;
    procedure Init; override;
    procedure Deinit; override;
    procedure MoveBegin(IsPass: boolean); override;
    function Moving: TMove; override;
    procedure MoveEnd; override;
  end;

implementation

// konstruktor
constructor TPlayerHuman.Create(PlayerNumber: integer);
begin
  OldMouseX := -1;
  OldMouseY := -1;
end;

// začátek tahu - budeme sledovat pohyb myši a tlačítka myši
procedure TPlayerHuman.MoveBegin(IsPass: boolean);
begin
  if Pass then
    Exit;
  DrawBoard.CaptureMouse(@MouseMove, @MouseClick);
end;

// tah hráče
function TPlayerHuman.Moving: TMove;
begin
  Clicked := False;
  // smyčka dokud hráč nestiskne tlačítko myši nebo není hra přerušená
  repeat
    Application.ProcessMessages;
    Sleep(1);
    if Game.Abort > abNoAbort then
      Exit;
  until Clicked;
  // vrať tah tam, kde se nachází kurzor myši
  Result := CreateMove(OldMouseX, OldMouseY, Number);
```

```
end;

// konec tahu hráče
procedure TPlayerHuman.MoveEnd;
begin
  // vypnutí zvýraznění políčka, kde se nachází myš
  if OldMouseX <> -1 then begin
    Game.Overlay[OldMouseX, OldMouseY] :=
      Game.Overlay[OldMouseX, OldMouseY] - [boHighlight1, boHighlight2];
    DrawBoard.DrawField(OldMouseX, OldMouseY);
  end;
  // přestaneme sledovat pohyb myši a tlačítka myši
  DrawBoard.ReleaseMouse;
end;

// událost pohybu myši po formuláři
procedure TPlayerHuman.MouseMove(Sender: TObject;
  Shift: TShiftState; x, y: integer);
var
  mx, my: integer;
begin
  // pokud se myš pohybuje nad nějakým políčkem hrací desky
  if (x > DrawBoard.GetScreenX(0)) and
    (x < DrawBoard.GetScreenX(BoardSize)) and
    (y > DrawBoard.GetScreenY(0)) and
    (y < DrawBoard.GetScreenY(BoardSize)) then begin
    mx := DrawBoard.GetBoardX(x);
    my := DrawBoard.GetBoardY(y);
    // a není stejné jako naposled
    if (mx <> OldMouseX) or (my <> OldMouseY) then begin
      // pak vypni staré zvýraznění políčka
      if OldMouseX <> -1 then begin
        Game.Overlay[OldMouseX, OldMouseY] :=
          Game.Overlay[OldMouseX, OldMouseY] -
          [boHighlight1, boHighlight2];
        DrawBoard.DrawField(OldMouseX, OldMouseY);
      end;
      // a nastav nové
      if Number = 1 then
        Game.Overlay[mx, my] := Game.Overlay[mx, my] + [boHighlight1]
      else
        Game.Overlay[mx, my] := Game.Overlay[mx, my] + [boHighlight2];
      DrawBoard.DrawField(mx, my);
      // ulož nové hodnoty polohy myši
      OldMouseX := mx;
      OldMouseY := my;
    end;
  end
  // pokud jsme zajeli kurzorem myši mimo hrací plochu
```

```
  else begin
    // pak vypni zvýraznění políčka na hrací desce
    if OldMouseX <> -1 then begin
      Game.Overlay[OldMouseX, OldMouseY] :=
        Game.Overlay[OldMouseX, OldMouseY] -
        [boHighlight1, boHighlight2];
      DrawBoard.DrawField(OldMouseX, OldMouseY);
    end;
    OldMouseX := -1;
    OldMouseY := -1;
  end;
end;

// událost stisknutí tlačítka myši
procedure TPlayerHuman.MouseClick(Sender: TObject);
begin
  Clicked := True;
end;

end.
```

B.12 UnitPlayerNetwork.pas

```
unit UnitPlayerNetwork;

interface

type
  // třída
  TPlayerNetwork = class(TPlayer)
  private
    // komponenta protokolu TCP z jednotky lNet
    TCP: TLTcp;
    // řetězec přijatého tahu
    ReceivedMove: string;
    procedure OnError(const msg: string; aSocket: TLSocket);
    procedure OnReceive(aSocket: TLSocket);
    procedure OnDisconnect(aSocket: TLSocket);
  public
    constructor Create(PlayerNumber: integer); reintroduce;
    procedure Init; override;
    procedure Deinit; override;
    destructor Destroy; override;
    procedure MoveBegin(IsPass: boolean); override;
    function Moving: TMove; override;
    procedure MoveEnd; override;
  end;
```

```
implementation

// pokud nastala chyba (spojení), přeruš hru
procedure TPlayerNetwork.OnError(const msg: string; aSocket: TLSocket);
begin
  if Game.Abort = abNoAbort then
    Game.Abort := abDisconnected;
end;

// přijali jsme zprávu ze sítě
procedure TPlayerNetwork.OnReceive(aSocket: TLSocket);
var
  s: string;
begin
  aSocket.GetMessage(s);
  if Length(s) = 0 then
    Exit;
  case s[1] of
    // M - řídící znak pro tah, další dva znaky jsou sloupec a řádek tahu
    'M':
      ReceivedMove := Copy(s, 2, 2);
  end;
end;

// pokud došlo k odpojení, přeruš hru
procedure TPlayerNetwork.OnDisconnect(aSocket: TLSocket);
begin
  if Game.Abort = abNoAbort then
    Game.Abort := abDisconnected;
end;

// konstruktor - vytvoří komponentu TCP
constructor TPlayerNetwork.Create(PlayerNumber: integer);
begin
  TCP := TLTcp.Create(nil);
  TCP.Timeout := 100;
  TCP.ReuseAddress := True;
end;

// inicializace hráče - zobrazí formulář připojení a čeká na spojení
procedure TPlayerNetwork.Init;
begin
  // předá kontrolu nad spojením formuláři připojení
  FormConnection.TCP := TCP;
  FormConnection.Number := Number;
  TCP.OnError := @FormConnection.OnError;
  TCP.OnReceive := @FormConnection.OnReceive;
  TCP.OnAccept := @FormConnection.OnAccept;
```

```
  FormConnection.Show;
  // čeká dokud není spojení navázáno nebo dokud není přerušena hra
  repeat
    TCP.CallAction;
    Application.ProcessMessages;
    if not FormConnection.Visible then
      Game.Abort := abDisconnected;
  until FormConnection.Connected or (Game.Abort > abNoAbort);
  FormConnection.Close;
  // vrací kontrolu nad spojením do této třídy
  TCP.OnError := @OnError;
  TCP.OnReceive := @OnReceive;
  TCP.OnDisconnect := @OnDisconnect;
end;

// deinicializace hráče, rozpojí TCP spojení
procedure TPlayerNetwork.Deinit;
begin
  TCP.Disconnect;
end;

// destruktor, zničí komponentu TCP
destructor TPlayerNetwork.Destroy;
begin
  TCP.Free;
end;

// začátek tahu hráče
procedure TPlayerNetwork.MoveBegin(IsPass: boolean);
begin
  // na začátku tahu pošleme připojenému hráči tah, který provedl hráč na
  // tomto počítači
  if Length(Game.History) > 0 then
    TCP.SendMessage('M' +
      Chr(Game.History[Length(Game.History) - 1].Move.x) +
      Chr(Game.History[Length(Game.History) - 1].Move.y));
end;

// tah hráče
function TPlayerNetwork.Moving: TMove;
begin
  ReceivedMove := '';
  // čekáme dokud neobdržíme tah od připojeného hráče nebo není
  // přerušená hra
  repeat
    Application.ProcessMessages;
    if Game.Abort > abNoAbort then
      Exit;
    TCP.CallAction;
```

```
  until ReceivedMove <> '';
  // dekódujeme tah ze zprávy
  Result := CreateMove(Ord(ReceivedMove[1]),
    Ord(ReceivedMove[2]), Number);
end;

end.
```

B.13 UnitFormConnection.pas

```
unit UnitFormConnection;

interface

type
  // třída formuláře
  TFormConnection = class(TForm)
    ButtonConnect: TButton;
    ButtonListen: TButton;
    ButtonCancel: TButton;
    GroupBoxConnect: TGroupBox;
    GroupBoxListen: TGroupBox;
    LabelText: TLabel;
    LabeledEditConnectAddress: TLabeledEdit;
    LabeledEditConnectPort: TLabeledEdit;
    LabeledEditListenPort: TLabeledEdit;
    procedure ButtonConnectClick(Sender: TObject);
    procedure ButtonListenClick(Sender: TObject);
    procedure ButtonCancelClick(Sender: TObject);
    procedure FormShow(Sender: TObject);
  public
    // komponenta TCP protokolu z jednotky lNet
    TCP: TLTcp;
    // číslo připojovaného hráče
    Number: integer;
    // spojení navázáno
    Connected: boolean;
    procedure OnError(const msg: string; aSocket: TLSocket);
    procedure OnReceive(aSocket: TLSocket);
    procedure OnAccept(aSocket: TLSocket);
  end;

var
  // instance formuláře
  FormConnection: TFormConnection;

implementation
```

```
// kliknutí na tlačítko "Připojit" - snaha připojit síťového hráče na
// daném portu
procedure TFormConnection.ButtonConnectClick(Sender: TObject);
var
  // port na kterém má připojovaný hráč čekat na spojení
  Port: integer;
begin
  // port získej z textového pole a otestuj, zda-li je správně zadán
  Port := StrToIntDef(LabeledEditConnectPort.Text, -1);
  if Port = -1 then begin
    LabelText.Caption := 'Chybně zadaný port';
    Exit;
  end;
  // pokus o spojení, nečeká se na výsledek, ten dorazí do OnReceive
  if TCP.Connect(LabeledEditConnectAddress.Text, Port) then
    LabelText.Caption := Format('Připojuji se na %s:%s',
      [LabeledEditConnectAddress.Text, LabeledEditConnectPort.Text])
  else
    LabelText.Caption := 'Nastala chyba, zkuste změnit port nebo adresu';
end;

// kliknutí na tlačítko "Čekat" - bude očekáváno připojení na daný port
procedure TFormConnection.ButtonListenClick(Sender: TObject);
var
  // port na kterém se poslouchá
  Port: integer;
begin
  // port získej z textového pole a otestuj, zda-li je správně zadán
  Port := StrToIntDef(LabeledEditListenPort.Text, -1);
  if Port = -1 then begin
    LabelText.Caption := 'Chybně zadaný port';
    Exit;
  end;
  // pokus se poslouchat na portu Port, nečeká se, konekce přijde do
  // OnAccept
  if TCP.Listen(Port) then
    LabelText.Caption := Format('Poslouchám na portu %s',
      [LabeledEditListenPort.Text])
  else
    LabelText.Caption := 'Nastala chyba, zkuste změnit port';
end;

// kliknutí na tlačítko "Storno" - zavření formuláře
procedure TFormConnection.ButtonCancelClick(Sender: TObject);
begin
  Close;
end;
```

```
// toto proběhne při zobrazení formuláře
procedure TFormConnection.FormShow(Sender: TObject);
begin
  LabelText.Caption := '';
  Connected := False;
end;

// nastala chyba v TCP
procedure TFormConnection.OnError(const msg: string; aSocket: TLSocket);
begin
  LabelText.Caption := 'Nelze se připojit, zkuste to znovu';
end;

// příjem dat, vyhodnocení
procedure TFormConnection.OnReceive(aSocket: TLSocket);
var
  s: string;
begin
  aSocket.GetMessage(s);
  if Length(s) = 0 then
    Exit;
  // testování prvního (řídícího znaku)
  case s[1] of
    // C - konekce přijata, jako parametr je číslo hráče na druhé straně,
    // je třeba zajistit, aby oba hráči nehráli za stejnou barvu
    'C':
      if Ord(s[2]) <> Number then begin
        Connected := True;
        TCP.SendMessage('C' + Chr(Number));
      end
      else begin
        LabelText.Caption := 'Nemůžete hrát za stejnou barvu';
        TCP.Disconnect;
      end;
  end;
end;

// připojení navázáno
procedure TFormConnection.OnAccept(aSocket: TLSocket);
begin
  // pošleme zprávu o připojení s číslem hráče, který je zde síťový
  TCP.SendMessage('C' + Chr(Number));
end;

end.
```

B.14 UnitPlayerCPU.pas

```
unit UnitPlayerCPU;

interface

type
  // třída
  TPlayerCPU = class(TPlayer)
    // časovač
    Timer: TTimer;
    // časovač doběhl
    TimerDone: boolean;
    procedure OnTimer(Sender: TObject);
  public
    constructor Create(ANumber: integer); override;
    procedure MoveBegin(aPass: boolean); override;
    procedure MoveEnd; override;
    destructor Destroy; override;
  end;

implementation

// událost časovače - časovač doběhl
procedure TPlayerCPU.OnTimer(Sender: TObject);
begin
  TimerDone := True;
end;

// konstruktor - založí časovač
constructor TPlayerCPU.Create(ANumber: integer);
begin
  Timer := TTimer.Create(nil);
  Timer.OnTimer := @OnTimer;
  Timer.Enabled := False;
  TimerDone := False;
end;

// začátek tahu hráče
procedure TPlayerCPU.MoveBegin(aPass: boolean);
begin
  // pokud hráč nemůže hrát, nic se nestane
  if Pass then
    Exit;
  // spuštění časovače
  TimerDone := False;
  Timer.Interval := CPUMinimumWait;
  Timer.Enabled := True;
```

```
end;

// konec tahu hráče
procedure TPlayerCPU.MoveEnd;
begin
  // čekání na časovač nebo přerušení hry
  repeat
    Sleep(1);
    Application.ProcessMessages;
  until TimerDone or (Game.Abort > abNoAbort);
  // vypnutí časovače
  Timer.Enabled := False;
end;

// destruktor - zničí časovač
destructor TPlayerCPU.Destroy;
begin
  Timer.Free;
end;

end.
```

B.15 UnitPlayerCPURandom.pas

```
unit UnitPlayerCPURandom;

interface

type
  // třída
  TPlayerCPURandom = class(TPlayerCPU)
  public
    constructor Create(PlayerNumber: integer); override;
    procedure Init; override;
    procedure Deinit; override;
    procedure MoveBegin(IsPass: boolean); override;
    function Moving: TMove; override;
    procedure MoveEnd; override;
    destructor Destroy; override;
  end;

implementation

// probíhá tah, toto je jediný nový kód třídy
function TPlayerCPURandom.Moving: TMove;
begin
  // vrací náhodný tah, o jeho legalitě se rozhoduje ve třídě hry
```

```
  Result := CreateMove(Random(BoardSize), Random(BoardSize), Number);
end;

end.
```

B.16 UnitPlayerCPUMinimax.pas

```
unit UnitPlayerCPUMinimax;

interface

type
  // třída
  TPlayerCPUMinimax = class(TPlayerCPU)
  private
    // třída pole hrací desky, kde probíhá ohodnocování budoucích možných
    // tahů v metodě Minimax
    Board: TBoard;
    // čítač iterací metody minimax
    RefreshCount: integer;
    // hloubka metody minimax pro tohoto hráče
    MinimaxDepth: integer;
    function Evaluation(MovingPlayer: byte): integer;
    function Minimax(Depth: integer; MovingPlayer: byte;
      Alpha, Beta: integer; WasPass: boolean = False): integer;
    function Winner(MovingPlayer: byte): integer;
  public
    constructor Create(PlayerNumber: integer;
      MaximumMinimaxDepth: integer); reintroduce;
    procedure Init; override;
    procedure Deinit; override;
    procedure MoveBegin(IsPass: boolean); override;
    function Moving: TMove; override;
    procedure MoveEnd; override;
    destructor Destroy; override;
  end;

implementation

// ačkoli se všude píše o minimaxu, ve skutečnosti se jedná o
// algoritmus negamax s alfa-beta prořezáváním
// Depth - aktuální hloubka (úroveň) stromu
// MovingPlayer - číslo hrajícího hráče na této úrovni
// Alpha, Beta - interval pro alfa-beta prořezávání
// WasPass - v předchozím pseudotahu, hráč nemohl zahrát tah
function TPlayerCPUMinimax.Minimax(Depth: integer;
  MovingPlayer: byte; Alpha, Beta: integer;
```

```
  WasPass: boolean = False): integer;
var
  // pole legálních tahů
  MoveArray: TMoveArray;
  i: integer;
  Value: integer;
  BestValue: integer = -MaxInt;
  // uložený stav pole hrací desky na této úrovni
  SaveBoard: TBoardArray;
  LocalAlpha: integer;
begin
  Inc(RefreshCount);
  // po každých sto iteracích proveď obsluhu formuláře a aplikace, aby
  // se s formulářem dalo stále pracovat i během výpočtu;
  // hodnota 100 je zase nastavena proto, aby se algoritmus zbytečně
  // nezpomaloval touto obsluhou
  if RefreshCount = 100 then begin
    Application.ProcessMessages;
    RefreshCount := 0;
  end;
  LocalAlpha := Alpha;
  // pokud jsme v konečné hloubce, pak rovnou vrať ohodnocení tohoto
  // stavu
  if Depth = 0 then
    Exit(Evaluation(MovingPlayer));
  MoveArray := Board.FindPossibleMoves(MovingPlayer);
  // pokud ani tento hráč nemůže hrát, pak je konec hry a vrací se
  // hodnota výhry nebo prohry
  if (Length(MoveArray) = 0) and WasPass then
    Exit(Winner(MovingPlayer) * ValueWin);
  // pokud hráč nemůže hrát, pak se vrací hodnota nemožnosti hrát +
  // hodnota dalších tahů
  if Length(MoveArray) = 0 then
    Exit(-ValueNoMove - Minimax(Depth - 1, AnotherPlayer(MovingPlayer),
      -Beta, -LocalAlpha, True));
  // uložení stavu pole hrací desky
  SaveBoard := Board.Board;
  // projedeme všechny legální tahy
  for i := 0 to Length(MoveArray) - 1 do begin
    // uskutečníme tah
    Board.DoMove(MoveArray[i]);
    // a pokud nebyla přerušena hra, spustíme minimax v další úrovni
    if Game.Abort = abNoAbort then
      Value := -Minimax(Depth - 1, AnotherPlayer(MovingPlayer),
        -Beta, -LocalAlpha);
    // vrátíme stav pole hrací desky
    Board.Board := SaveBoard;
    // vybereme lepší ohodnocení
    BestValue := Max(Value, BestValue);
```

```
    // alfa-beta prořezávání
    if BestValue >= Beta then
      break;
    if BestValue > LocalAlpha then
      LocalAlpha := BestValue;
  end;
  Result := BestValue;
end;

// funkce vrací 1 pokud daný hráč MovingPlayer vyhrál nebo remizoval,
// jinak vrací -1
function TPlayerCPUMinimax.Winner(MovingPlayer: byte): integer;
var
  i, j: integer;
  Count: array[1..2] of integer;
begin
  Count[1] := 0;
  Count[2] := 0;
  // vypočti skóre
  for j := 0 to BoardSize - 1 do
    for i := 0 to BoardSize - 1 do
      if Board.Board[i, j] > btFree then
        Inc(Count[Board.Board[i, j]]);
  // porovnání skóre
  if Count[MovingPlayer] >= Count[AnotherPlayer(MovingPlayer)] then
    Exit(1)
  else
    Exit(-1);
end;

// ohodnocující funkce z pohledu hráče MovingPlayer
function TPlayerCPUMinimax.Evaluation(MovingPlayer: byte): integer;
var
  i, j: integer;
  Count: array[1..2] of integer;
begin
  // prostý počet políček
  Count[1] := 0;
  Count[2] := 0;
  for j := 0 to BoardSize - 1 do
    for i := 0 to BoardSize - 1 do
      if Board.Board[i, j] > btFree then
        Inc(Count[Board.Board[i, j]]);
  Result := ValueField * (Count[MovingPlayer] -
    Count[AnotherPlayer(MovingPlayer)]);
  // rohové pozice
  Count[1] := 0;
  Count[2] := 0;
  for j := 0 to 1 do
```

```
    for i := 0 to 1 do
      if Board.Board[i * (BoardSize - 1), j * (BoardSize - 1)] >
        btFree then
        Inc(Count[Board.Board[i * (BoardSize - 1), j *
          (BoardSize - 1)]]);
  Result := Result + ValueCorner * (Count[MovingPlayer] -
    Count[AnotherPlayer(MovingPlayer)]);
  // pozice na hranách
  Count[1] := 0;
  Count[2] := 0;
  for i := 1 to BoardSize - 2 do
    for j := 0 to 1 do begin
      if Board.Board[i, j * (BoardSize - 1)] > btFree then
        Inc(Count[Board.Board[i, j * (BoardSize - 1)]]);
      if Board.Board[j * (BoardSize - 1), i] > btFree then
        Inc(Count[Board.Board[j * (BoardSize - 1), i]]);
    end;
  // výpočet výsledné hodnoty
  Result := Result + ValueEdge * (Count[MovingPlayer] -
    Count[AnotherPlayer(MovingPlayer)]);
end;

// kostruktor - navíc založíme pole hrací desky pro potřeby minimaxu
constructor TPlayerCPUMinimax.Create(PlayerNumber: integer;
  MaximumMinimaxDepth: integer);
begin
  Board := TBoard.Create;
  MinimaxDepth := MaximumMinimaxDepth;
end;

// tah začíná - navíc uložíme pole hrací desky ze hry do našeho
// lokálního pole
procedure TPlayerCPUMinimax.MoveBegin(IsPass: boolean);
begin
  if Pass then
    Exit;
  Board.Board := Game.Board.Board;
end;

// probíhá tah
function TPlayerCPUMinimax.Moving: TMove;
var
  MoveArray: TMoveArray;
  BestMoves: TMoveArray;
  MaxScore: integer = -MaxInt;
  CurrentScore: integer;
  i: integer;
begin
  // nalezneme legální tahy
```

```
  MoveArray := Board.FindPossibleMoves(Number);
  RefreshCount := 0;
  // všechny je projdeme
  for i := 0 to Length(MoveArray) - 1 do begin
    // provedení tahu (pouze v lokálním poli)
    Board.DoMove(MoveArray[i]);
    // ohodnotíme tento tah minimaxem
    CurrentScore := -Minimax(MinimaxDepth, AnotherPlayer(Number),
      -MaxInt, -MaxScore);
    // test přerušení
    if Game.Abort > abNoAbort then
      Exit;
    // pokud máme lepší ohodnocení, vymaž stávající pole nejlepších tahů
    // a přidej tam tento
    if CurrentScore > MaxScore then begin
      MaxScore := CurrentScore;
      SetLength(BestMoves, 1);
      BestMoves[0] := MoveArray[i];
    end
    else
    // nebo pokud je ohodnocení stejné, přidej tento tah do pole
    // nejlepších tahů
    if CurrentScore = MaxScore then begin
      SetLength(BestMoves, Length(BestMoves) + 1);
      BestMoves[Length(BestMoves) - 1] := MoveArray[i];
    end;
    // vrať lokální pole hrací desky do stavu, který je ve hře
    Board.Board := Game.Board.Board;
  end;
  // pokud máme nastavený náhodný výběr nejlepšího tahu, pak to udělej
  if RandomBestMoves then
    Result := BestMoves[Random(Length(BestMoves))]
  else
    // jinak vrať první nalezený tah
    Result := BestMoves[0];
end;

// destruktor zničí lokální pole hrací desky
destructor TPlayerCPUMinimax.Destroy;
begin
  Board.Free;
end;

end.
```