VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

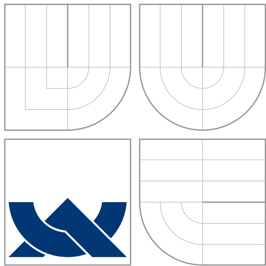
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

HRA GOMOKU

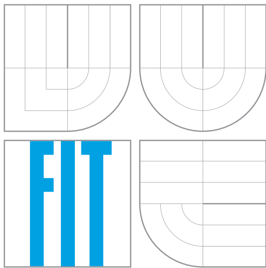
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE MARTIN BASOVNÍK
AUTHOR

BRNO 2011



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

HRA GOMOKU

GOMOKU GAME

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE MARTIN BASOVNÍK
AUTHOR

VEDOUCÍ PRÁCE Ing. JAN KNĚŽÍK
SUPERVISOR

BRNO 2011

Abstrakt
Tato bakalářská práce je zaměřena na vývoj umělé inteligence pro deskovou hru gomoku. Nejdříve popisuje potřebný teoretický základ pro vývoj inteligentního protihráče. Dále uvažuje nad možnou paralelizací výpočtů a algoritmy, které se k tomu vztahují. Následně je popsána samotná implementace hry gomoku. V závěrečné části je zdokumentováno testování aplikace, zveřejňují se výsledky vývoje a nastiňují se případná další rozšíření aplikace.

Abstract
This bachelor's thesis is focused on the development of artificial intelligence for board game Gomoku. Firstly it describes the necessary theoretical basis for the development of intelligent opponents. Secondly it considers the possibility of parallelization. Next it describes the implementation of the game Gomoku. Finally it documents testing process, publishes the results of the development, and outlines possible further extensions to the application.

Klíčová slova
gomoku, piškvorky, deskové hry, umělá inteligence, paralelismus

Keywords
gomoku, board games, artificial intelligence, parallelism

Citace
Martin Basovník: Hra Gomoku, bakalářská práce, Brno, FIT VUT v Brně, 2011

Hra Gomoku

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Jana Kněžíka

....
Martin Basovník
10. května 2011

Poděkování

Tímto děkuji Ing. Janu Kněžíkovi za vedení a podporu při vzniku bakalářské práce, za poskytnutí konzultací, podnětných a cenných rad, bez nichž by práce nemohla vzniknout.

© Martin Basovník, 2011.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

Úvod **3**

1 Hra Gomoku **4**
1.1 Historie hry 4
1.2 Oficiální pravidla hry 5

2 Inteligentní počítačový soupeř **6**
2.1 Ohodnocování stavu hry 6
2.2 Algoritmy pro prohledávání stavového prostoru 7
2.2.1 Algoritmus MiniMax 7
2.2.2 Alfa-beta prořezávání 8
2.2.3 Algoritmus Negamax 9
2.2.4 Algoritmus NegaScout (Principal Variation Search) 9
2.3 Další metody pro zrychlení prohledávání stavového prostoru 10
2.3.1 Databáze tahů 10
2.3.2 Vyčlenění některých uzlů z prohledávání 10
2.4 Zdokonalování umělé inteligence učením 11

3 Paralelní prohledávání stavového prostoru **12**
3.1 Amdahlův zákon 12
3.2 Gustafsonův zákon 13
3.3 Synchronizace paralelního prohledávání 13
3.4 Paralelní algoritmy pro prohledávání stavového prostoru 14
3.4.1 Principal Variation Splitting (PVSplit) 14
3.4.2 Young Brothers Wait Concept (YBWC) 15
3.4.3 Dynamic Tree Splitting (DTS) 16

4 Realizace aplikace **17**
4.1 Návrh objektového modelu 17
4.1.1 Třída Game 17
4.1.2 Třída Field 18
4.1.3 Třída Box 18
4.1.4 Třída Player 19
4.1.5 Třída Brain 19
4.2 Návrh umělé inteligence 20
4.2.1 Ohodnocovací funkce stavu hry 20
4.2.2 Prohledávání stavového prostoru 21
4.2.3 Paralelizace algoritmů 22

4.3 Grafické rozhraní 23
4.3.1 Ovládání aplikace 24

5 Dosažené výsledky **25**
5.1 Testování síly umělé inteligence 25
5.1.1 Pisky 1.3 (konkurenční aplikace) 25
5.1.2 Piškvorky 8.4 (konkurenční aplikace) 26
5.1.3 Testování hry s profesionálním hráčem 27
5.2 Výsledky paralelizace umělé inteligence 28

6 Závěr **30**

A Obsah DVD **32**
A.1 Spustitelný program 32
A.2 Zdrojové soubory 32

B Zkoumané herní situace **33**

Úvod

Hry jsou odjakživa nedílnou součástí trávení volného času. Každé dítě miluje hry a dokáže si hrát od rána do večera. To ale neznamená, že by dospělí lidé hry nehráli. Většinou se ale zajímají spíše o společenské hry nebo o nějaké logické hry, u kterých se musí přemýšlet. Mezi takovéto logické hry patří určitě šachy, sudoku nebo například hra gomoku. Právě o hře gomoku pojednává tato práce.

V kapitole 1 se hlouběji seznámíte s pravidly hry Gomoku a zjistíte, že se trochu liší od klasických pravidel, které asi zná každý ze školních lavic. Dozvíte se, proč tyto změny v pravidlech nastaly a jaká pravidla mají dnešní oficiální turnaje. Dále se v této kapitole dozvíte něco o historii hry Gomoku. Hra vznikla již před několika tisíci lety, ale zajímavé je, že na několika místech najednou.

Kapitola 2 pojednává o principech vývoje inteligentního soupeře. Dozvíte se zde o tom, jak počítač dokáže vyhodnotit, zda je pro něj aktuální herní pozice výhodná nebo ne. Dále se zde dozvíte, jak se dá docílit toho, aby počítač přemýšlel několik tahů dopředu. Zajímavé na tom je, že počítač musí v přemýšlení dopředu hrát i za Vás a musí předpokládat, že zahrajete vždy ten nejlepší možný tah. Na základě tohoto tahu pak vyhodnotí nejlepší možnou reakci. Kapitola také pojednává o různých metodách, jak výpočet urychlit.

Kapitola 3 se zabývá další možností urychlení výpočtu, a to paralelizací. Dozvíte se zde, jak je možné do výpočtu zapojit více procesorů a jaké úskalí to za cenu zrychlení přináší. Pokud daný problém totiž počítají dva procesory, tak to ještě neznamená, že výpočet bude dvakrát rychlejší. V kapitole jsou vysvětleny základní mechanismy, jak je možné paralelismus naprogramovat.

Kapitola 4 se v první části zabývá objektovým modelem aplikace a vysvětluje vztahy jednotlivých tříd. V další části se zabývá realizací umělé inteligence s vybranými algoritmy.

Kapitola 5 se v první části zabývá testováním herní sily aplikace. Obsahuje několik záznamů partií s konkurenčními aplikacemi. Dále kapitola obsahuje zhodnocení úspěšnosti paralelizace výpočtů.

Závěrečná kapitola shrunuje celou práci. Rekapituluje implementované mechanismy, zhodnocuje herní sílu aplikace a doporučuje případná další rozšíření.

Kapitola 1

Hra Gomoku

Gomoku je velice stará strategiská hra pro dva hráče. Hraje se na speciální desce zvané *Goban* s černými a bílými kameny. Goban je deska, která vznikla původně pro hru Go, ale dá se na ní hrát množství dalších her. Původně byl goban rozdělen na 18×18 čtvercových políček a kameny se pokládaly na jejich průsečíky. Celkem bylo tedy 19×19 pozic na položení kamenu. V dnešní době se ale hraje na upravené hrací ploše, která má 14×14 čtverců, tedy 15×15 pozic. Při hře gomoku se hráči střídají v pokládání kamenů (každý má jednu barvu, začínající hráč černou) a vyhrává ten hráč, kterému se podaří vytvořit řadu pěti kamenů v příčném nebo v diagonálním směru. Evropskou obdobou této hry je hra Piškvorky. Hlavní rozdíl mezi těmito hrami je ale takový, že Gomoku je hra prostorově omezená a v oficiálních pravidlech nepovoluje vítězství při vytvoření delší řady než pěti kamenů.

1.1 Historie hry

První zmínky o této hře se objevili už 2000 let př. Kr. v oblasti dnešní Činy, takže gomoku je více než 4000 let stará hra. Nezávisle na Číně byly údaje o této hře objeveny dokonce i v antickém Řecku a v předkolumbovské Americe. Usuzuje se, že tato hra vznikla z ještě starší hry Go, která se objevila také nejprve v oblastni Číny a Japonska. Slovo gomoku pochází z Japonštiny a slává se ze dvou slov. Slovo „go“ znamena zpět a „moku“ průsečík.

Postupem času se úroveň hráčů hry Gomoku velice zdokonalila a pravidla hry znamenaly obrovskou výhodu pro začínajícího hráče. Důsledek byl takový, že dva silní hráči hráli stále remízů, tedy každý hráč vyhrál při svém začátku. Japonci už více než před sto lety v praxi dokázali, že začínající hráč vždy vyhraje. Matematicky to potom v minulém století dokázal holanďan Victor Allis. Zjistil, že začínající hráč nemůže prohrát. V gomoku tedy nelze prohrát, lze jen špatně zahrát zahájení. Proto se začali objevovat změny oproti původním pravidlům, aby se šance hráčů na výhru srovnaly. Jednou ze změn bylo vytvoření zakázaných tahů tzv. faulů. Pokud například začínající černý hrač vytvoří kombinaci 3x3 kamenů (i když by se bránil) automaticky prohrává. Dalším faulem bylo například vytvoření pravidla, že jedním tahem se nesmí vytvořit kombinace kamenů 4x3x3, přitom 5x3x3 bylo povoleno. Mezi další změny patřilo omezení velikosti hrací plochy z 19x19 na 15x15 a odepření výhry černého, pokud udělá přesah šesti a více kameny v jeho barvě. Změn pravidel bylo mnoho, ale ne všechny patří mezi pravidla oficiální federace piškvorek a renju [10].

1.2 Oficiální pravidla hry

Dle rozmanitosti doplňujících pravidel, které se snaží smazat výhodu začínajícího, má tedy gomoku několik variant. Pravděpodobně nejvyrovnanější a také oficiální varianta pro internetové i živé turnaje je varianta s názvem SWAP2. Začínající hráč postaví pozici prvních tří kamenů (tedy černý, bílý, černý) kdekoli na hrací desce a druhý hráč si posléze vybere barvu, za kterou bude hrát. Možnost navrhnout otevírací postavení je sice výhodou, avšak hráč musí volit takové otevření, které umí zahrát za obě strany. Druhý hráč, který si vybírá barvu, má ještě možnost postavit další dva kameny a výběr barvy je zpět na začínajícím hráči. Dále hra pokračuje klasicky s tím omezením, že hráč vyhráva při vytvoření řady kamenů v jeho barvě o právě pěti kamenech, ne více. Hrací plocha (převážně 15×15) je omezena pevně stanoveným okrajem. Pomyslná pole za hracím okrajem nemohou být nijak použita. [9].

Kapitola 2

Inteligentní počítačový soupeř

Než se vynalezly první počítače, tak se hra gomoku dala hrát pouze mezi dvěma hráči (člověk – člověk). S rozvojem počítačů a oboru umělá inteligence jsme dospěli do stavu, kdy můžeme naprogramovat inteligentního počítačového soupeře a hrát s ním (člověk – počítač). Jaké výhody a nevýhody z hlediska úrovně hry má hra s takovýmto počítačovým soupeřem? To je samozřejmě závislé na úrovni jeho umělé inteligence. Jedna ze základních výhod dobře naprogramovaného protihráče by měla být vlastnost neudělat chybu z nepozornosti, například nevšimnout si, že má hráč řadu již tří po sobě jdoucí kameny a chystá se na čtvrtý, což může být u hry s lidským faktorem vcelku běžnou záležitostí, zvlášť když je hra časově omezená. Klíčovou problematikou při implementaci inteligentního protihráče je zjistit, kterým tahem se hráč dostane do nejlepšího stavu hry, zjednodušeně řečeno, který tah je nejvýhodnější. Proto si počítač hodnotí jednotlivé možné stavy hry a následně udělá tah do toho nejvýhodnějšího. Jak ale stavy hry ohodnotit?

2.1 Ohodnocování stavu hry

Hra Gomoku patří do her, kde mají oba hráči absolutní přehled nad aktuálním stavem hry. Vidí všechny kameny a nic jim není zatajeno. Jak ale efektivně ohodnotit jednotlivé možné tahy? To je základní otázka při implementaci umělých inteligencí systémů. Počítačový protihráč může hrát právě tak dobře, jak je dobrá jeho ohodnocovací funkce stavu hry.

Počítač si tedy ohodnotí všechny jeho možné tahy v aktuálním stavu hry. K tomu existuje spousta metod. Jedna z velice známých a používaných metod je vyhledávání výhodných vzorů na hrací desce. Když například počítač najde tři po sobě jdoucí kameny v jeho barvě a nejsou ohraničeny cizím kamenem nebo krajem hrací desky, tak pozice rozšiřující tři kameny na čtyři bude zřejmě velmi dobře ohodnocena. Počítač takto může vyhledávat dvojice, ohraničené dvojice z jedné strany, trojice, ohraničené trojice z jedné strany, čtveřice a další kombinace kamenů ve stejné barvě a zjišťuje, které kombinace by přidáním daného kamene vznikly. Každá kombinace je jinak ohodnocena.

Jednou z dalších metod je sčítání kamenů ve všech směrech do určité vzdálenosti (většinou počet kamenů ve vítězné kombinaci, tedy 5) od ohodnocovaného pole. Pro konkrétního hráče se ohodnotí políčko tak, že každý vlastní soused zvyšuje ohodnocení a nepřátelský kámen ohodnocení může například nulovat. Je potřeba zde zohlednit větší výhodnost kamenů v řadě než roztroušenost kamenů v různých směrech. Ohodnocení všech políček se pak může sečíst a dostaneme aktuální ohodnocení stavu hry pro zvoleného hráče.

Dále může počítač ohodnocovat počet možných vítězných kombinací ve zvoleném prostoru a porovnávat to s počtem možných vítězných kombinací protihráče. Myšleno tak, že počítač zkoumá, na kolik způsobů může do zvoleného prostoru naskládat pět kamenů za sebou v porovnání se soupeřem. Takových metod lze vymyslet mnoho! Liší se samozřejmě kvalitou, ale i časem výpočtu, což je také velice důležitý faktor, ke kterému se vrátím později. Občas je proto výhodnější zvolit i jednoduchou, relativně kvalitní metodu, než nějakou super metodu, která je tak náročná, že se s ní už rozumně nedá pracovat v prohledávání *stavového prostoru.* Důležitým předpokladem pro předcházející metody je fakt, že je potřeba posuzovat stav hry z hlediska obou hráčů. Pokud tedy ohodnocuji hru pro hráče na tahu, tak ohodnotím kladně výhodné pozice pro něj a záporně výhodné pozice pro protihráče. Obě ohodnocení pak nakonec od sebe odečtu a kladný výsledek značí spíše dobrý stav pro hráče na tahu, naopak záporný výsledek značí dobrý stav spíše pro soupeře.

Nejlépe vyhodnocený tah v daném kole nemusí však být nejlepším tahem vedoucí k vítězství! K tomu by bylo dobré si představit, jak na daný tah zareaguje protihráč, jaká bude odpověď atd. . . Jedná se o již zmíněné prohledávání stavového prostoru.

2.2 Algoritmy pro prohledávání stavového prostoru

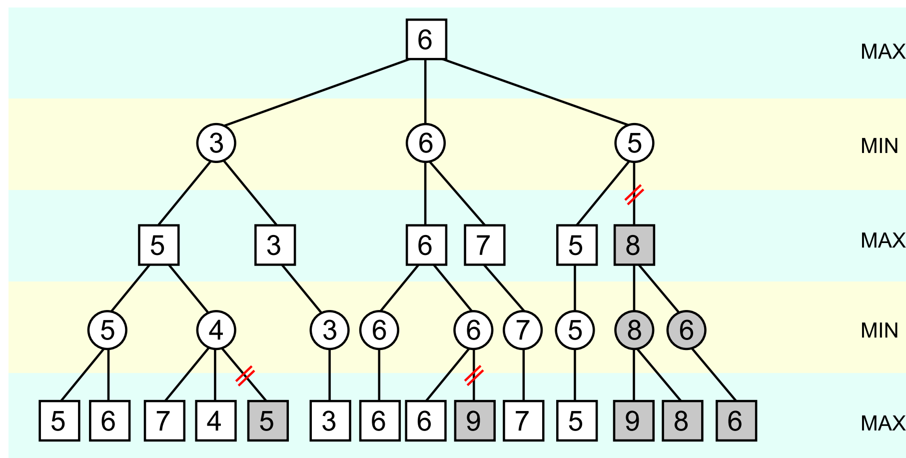
Stavový prostor je reprezentován stromovou strukturou. Kořen stromu představuje aktuální stav hry a každá další úroveň představuje jeden další tzv. půltah (tah jednoho nebo druhého hráče). Je zřejmé, že strom se exponenciálně zvětšuje. Při hrací desce 15×15 se totiž může jako první tah vybrat jeden z 225 tahů, na které protihráč odpoví jedním z 224 možných tahů, a již nyní má náš stavový prostor 50400 stavů. Pro výpočet celkového počtu stavů se dá použít matematická funkce faktoriál.

Kdybychom měli dobrou ohodnocovací funkci, stačili jsme prozkoumat celý stavový prostor a byli jsme hráči černé barvy, tak bychom našli vždy vítěznou cestu (vysvětleno v kapitole 1.1). Pokud bychom ale opravdu chtěli prohledávat celý stavový prostor u hry Gomoku, tak by to při současných počítačových výkonech mohlo trvat desítky let. Proto se většinou omezuje stavový prostor prohledávání na určitý počet úrovní, do které se má algoritmus prohledávání zanořit. Základním algoritmem prohledávání stavového prostoru je tzv. *MiniMax.*

2.2.1 Algoritmus MiniMax

Tento základní algoritmus pracuje na principu prohledávání stavového prostoru do hloubky s omezením hloubky prohledávání. Algoritmus projde všechny možné tahy, provede ohodnocení vzniklých pozic a vybere ten tah, který přinese nejvýhodnější pozici. Ohodnocení se provede buď přímo pomocí statické ohodnocovací funkce (pokud se nacházíme v konečném stavu stromu nebo v maximálním povoleném zanoření) nebo rekurzivním voláním téhož algoritmu za soupeře. Počet rekurzí, jak již bylo zmíněno výše, je omezené. Metoda vrací ohodnocení nejvýhodnějšího možného tahu pro hráče, který je na řadě a ukazatel na daný tah. Metoda se nazývá MiniMax, protože pro hráče, který je na tahu vybírá maximální možné ohodnocení tahu v lichých úrovních a při tahu protihráče zase vybírá minimální ohodnocení tahu v sudých úrovních. To představuje střídání hráčů ve hře. Protihráč vybírá minimální možné ohodnocení, protože si jistě nepřeje, aby jeho oponent vyhrál.

Algoritmus MiniMax není ve své základní verzi příliš používán, protože má jeden velký nedostatek. Prohledává úplně celý stavový prostor. Časová náročnost s každou úrovní zanoření roste exponenciálně. Z této metody vychází prakticky všechny další metody prohle-



Obrázek 2.1: Příklad $\alpha/\beta-$prořezávání [4].

dávání, ale vždy se snaží problém s velkým počtem uzlů ve stavovém prostoru nějak řešit. Základním rozšířením je princip *Alfa-Beta prořezávání*.

2.2.2 Alfa-beta prořezávání

Alfa-Beta prořezávání je doplňkový kód MiniMaxu, který výrazně zlepšuje jeho funkcionalitu. Metoda je založena na pozorování, že pokud právě zpracovávaný půltah už nemůže obstát v konkurenci s jiným, nemusíme dál prohledávat jeho důsledky. Tato metoda v optimálních případech zrychluje běh prohledávání až $2\times$ [7]. Účinnost metody lze zvýšit vhodnou volbou pořadí vyhodnocování jednotlivých větví. Vhodné je prohledávat větve, které by měly obsahovat lepší výsledky, protože tento fakt zvyšuje pravděpodobnost ořezání horší podvětve. Pokud bychom začali prohledávat od nejhorších větví po nejlepší, tak by se v samém důsledku nemuselo ořezat nic a průběh by byl stejný jako u metody MiniMax. Tato metoda využívá dvě mezní hodnoty:

- $\alpha-$Reprezentuje dolní mez ohodnocení uzlu. Na začátku nabývá hodnoty $-\infty$. Aktualizuje se v MAX úrovních při nalezení většího ohodnocení.
- $\beta-$Reprezentuje horní mez ohodnocení uzlu. Na začátku nabývá hodnoty $+\infty$. Aktualizuje se v MIN úrovních při nalezení menšího ohodnocení.

Při dosažení podmínky $\alpha \geq \beta$ dojde k ořezání zbytku větve [6].

Na obrázku 2.1 je vidět příklad vyhledávání metodou Alfa-Beta. Úroveň MAX značí hráče, který je řadě a úroveň MIN značí protihráče. MAX vybírá maximální ohodnocení a MIN minimální (pro něj výhodné). V nejpravější větvi nemusí být podvětev začínající ohodnocením 8 vůbec prohledána, protože předchozí větev vrátila 5 a v dané úrovni se vybírá minimum. Neprohledávaná větev by mohla ohodnocení MIN leda zmenšit, ale na výběru druhé větve s ohodnocením 6, když vybíráme MAX v nulté úrovni, už to nehraje roli. Nejvýhodnější tah pro hráče, který je na řadě, by bylo jít prostřední větví, tedy na uzel s ohodnocením 6 v první úrovni stromu.

2.2.3 Algoritmus Negamax

Algoritmus Negamax je vlatně pouze variace zápisu algoritmu Mini-Max. Mini-Max má spoustu kódu, který je na hranici duplicity. Je to způsobeno téměř stejným vyhodnocováním hráče MIN a MAX. Negamax na základě plátnosti rovnice 2.1, používá společný kód pro MIN a MAX část z klasivkého MinMax algoritmu a dává nám pak lepší jádro, na které se váží pokročilejší algoritmy jako například *NegaScout*.

$$max(a, b) \quad = -min(-a, -b) \tag{2.1}$$

2.2.4 Algoritmus NegaScout (Principal Variation Search)

Algoritmus NegaScout je vylepšená varianta Negamaxu s α/β prořezáváním. Tento algoritmus nikdy nebude prohledávat uzel, který by byl prořezán α/β prořezáváním. Důležitým znakem tohoto algoritmu je skutečnost, že potomci uzlu se generují již v uspořádaném pořadí, čehož je dosáhnuto mělkým prohledáváním stavu hry v dané úrovni. Negascout proto prořezává prostor výrazně efektivněji. Algoritmus předpokládá, že první uzel v seznamu potomků je ten nejlepší, a tedy je součástí *Principal Variation*, což je nejlepší cesta k výhře. Ověří se to tak, že nastavíme tzv. *null window* (neboli *scout window*, tzn. $\alpha = \beta$), které je rychlejší než řádné $\alpha\beta$ okno. Pokud průzkum selže, tak uzel nebyl v Principal Variation a pokračuje se klasicky. Algoritmus má lepší výsledky než klasický NegaMax s α/β prořezáváním právě pokud procházíme uspořádaný seznam potomků dle nějaké heuristiky. Pokud má seznam náhodné uspořádání, tak bude NegaScout pomalejší ne kvůli tomu, že by prohledával více odlišných uzlů, ale spostu uzlů bude prohlédávat vícekrát, protože nebude zabírat metoda nulového okénka. Negascout obvykle dosahuje zrychlení až 10% [8].

Algoritmus 1: NegaScout

```
if node is a terminal node or depth=0 then
    return the heuristic value of node
endif
b := beta
foreach child of node do
    doMove(child)
    v := -negascout(child, depth-1, -b, -alpha)
    if alpha < v  < beta and not the first child then
        v := -negascout(child, depth-1, -beta, -v)
    endif
    undoMove()
    alpha := max(alpha, v)
    if alpha ≥ beta then
        return alpha
    endif
    beta := alpha+1
endfch
return alpha
```

2.3 Další metody pro zrychlení prohledávání stavového prostoru

Jak už bylo zmíněno výše, prohledávat celý stavový prostor je časově extrémně náročné. Proto se využívají metody s omezenou hloubkou prohledávání. Není ale úplně od věci tento prostor ještě trochu zmenšit. Existuje spousta metod, jak prohledávací prostor omezit nebo vymezit situace, kdy neprohledávat.

2.3.1 Databáze tahů

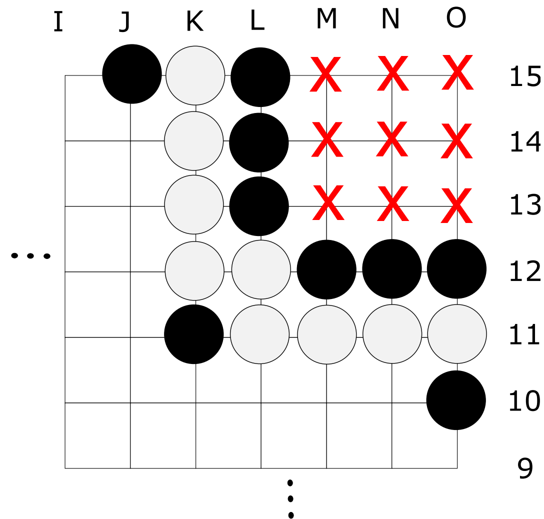
Určitě by bylo výhodné, kdybychom na každý tah znali předem nejlepší odpověď. Potom by byl počítač neporazitelný. Kdybychom si chtěli předpočítat všechny tahy a odpovědi na ně dopředu, tak by to bylo extrémně paměťově neekonomické a prakticky u většiny her nemožné. Není ale od věci si předpočítat odpověď na pár tahů dopředu při začátcích partie. Začátky se opakují a není tolik možností, jak efektivně začít. Proto může být předpočítání tahů na počátečních několik tahů hry vhodné a ušetříme tak náročnosti prohledávání. Musíme si dát ale pozor na to, zda jsou naše předpočítané tahy opravdu dobrým řešením. V opačném případě bychom už na začátku každé partie udělali chybu a začátek hry je opravdu zásadní moment.

Dále si představme situaci, že můžeme udělat v určité situaci dva výhodné tahy. Vybereme si jeden a prohledáváme stavový prostor pro další tah. Kdybychom si vybrali ten druhý, tak se bude hrací plocha lišit pouze v tomto kamenu a při prohledávání následujícího tahu by byl celý podstrom pro opačný tah. Dává tedy smysl, si uložit již řešené situace do nějaké transpoziční tabulky s vhodnou hashovací funkcí. V momentě kdy narazíme na situaci, kterou jsme již řešili, tak ji nebudeme procházet znova, ale použijeme již vypočtené ohodnocení.

2.3.2 Vyčlenění některých uzlů z prohledávání

Existuje pár situací kdy je vhodné určité tahy neprohledávat a ušetřit tak na časových nárocích. Můžeme například diskutovat o tom, zda má cenu prohledávat tahy, které se vyskytují od ostatních více než počet polí vedoucích k výhře. Čím vzdálenější pole od centra hry na hracím poli, tím menší ohodnocení by tento kámen dostal.

Další situace, kdy by bylo vhodné vyčlenit některé tahy z prohledávání je nemožnost jejich použití k výhře. Pokud máme na hrací ploše políčka, které jsou dosavadní hrou obskládaná tak, že už pomocí nich nelze vytvořit řady pěti stejných kamenů, tak je zbytečné podstromy prohledávat. Test na takováto *mrtvá pole* (viz. obrázek 2.2) bude nutné dělat na začátku každého tahu. Pokud bude testovací rutina efektivně napsaná, tak by se rozhodně měla vyplatit.



Obrázek 2.2: Obrázek znázorňuje pozice na hrací ploše, které jsou označovány jako *mrtvé*. Nemohou být již součástí výherní kombinace.

2.4 Zdokonalování umělé inteligence učením

Kdyby byl počítač jako člověk, tak by se stálým hraním a procvičováním neustále zdokonaloval. Člověk nabírá každou hrou zkušenosti a na základě těchto zkušeností se zlepšuje. Zlepšení se projeví v následujících hrách. Důležité je, abychom se ale učili od kvalitních hráčů. Právě hraní s kvalitními hráči nás posune dopředu. Jak ale zjistíme, zda konkrétní tah byl dobrý nebo ne? Špatný tah se může projevit třeba i po deseti tazích, tak jak poznáme, že právě ten tah byl špatný? Jediným objektivním kritériem je výhra/prohra. Pokud jsme partii vyhráli, máme relativní jistotu, že několik posledníh tahů jsme zahráli dobře. Pokud tedy implementujeme naši umělou inteligenci se schopností učení se z předchozích her, je důležité vyvozovat výsledky z co největší množiny her s co nejlepšími soupeři. Na základě výsledků pak můžeme měnit databázi počátečních tahů, ohodnocovací funkci a další věci.

Kapitola 3

Paralelní prohledávání stavového prostoru

V poslední době se stále více začínají rozmáhat víceprocesorové počítačové stanice nebo procesory s více jádry. Každý procesor potom může vykonávat zadané operace autonomně, nezávisle na dalším procesoru. V daných situacích nám nejde až tak o výkon jednotlivých procesorů, ale pouhá dekompozice problému na podproblémy a jejich nezávislé řešení může výrazně urychlit výpočet.

V teorii her můžeme víceprocesorové systémy použít k rychlejšímu prohledávání stavového prostoru. Použití N procesorů, ale určitě neznamená, že se $N\times$ zrychlí. Zaprvé není úplně jednoduché celý problém algoritmizovat tak, aby šel řešit paralelně, a proto se některé úseky kódu musejí řešit sekvenčně, a za druhé musíme zajistit určitou synchronizaci těchto paralelních výpočtů. Objasnění je v následujících odstavcích.

3.1 Amdahlův zákon

Amdahlův zákon omezuje maximální možné zrychlení paralelního výpočtu oproti sériovému, způsobenou nutností sériového zpracování v jinak paralelním algoritmu. Pokud f značí procento operací, které nelze paralelizovat a p značí počet procesorů, které máme k dispozici, tak maximální zrychlení lze výpočítat ze vzorce 3.2. Limita v nekonečnu ve vzorci 3.3 nám dokazuje, že zrychlení paralelního algoritmu může být maximálně $1/f$. Pokud se tedy například 90% kódu v algoritmu zpracovává paralelně a 10 sekvenčně, tak maximální možné zrychlení oproti sériovému výpočtu při 8 procesorech je pouze zhruba $4,7\times$. Ve vzorci 3.1 je definovaná proměnná f, kde P_s značí dobu běhu sériového výpočtu a P_p značí dobu běhu paralelního výpočtu. [1]

$$f = \frac{P_s(n)}{P_s(n) + P_p(n)} \tag{3.1}$$

$$S(n,p) \leq \frac{1}{f + \frac{1-f}{p}} \tag{3.2}$$

$$\lim_{p\to\infty} S(n,p) \leq \lim_{p\to\infty} \frac{1}{f + \frac{1-f}{p}} = \frac{1}{f} \tag{3.3}$$

3.2 Gustafsonův zákon

Amdahlův zákon docela rapidně omezuje maximální možné zrychlení výpočtu. Gustafson se na věc dívá trochu z jiného pohledu a říká, že s vyšším počtem procesorů je možné stejný problém vyřešit ve stejném čase přesněji. [2] Podobně jako v případě Amdahlova zákona s rostoucím počtem procesorů roste i zrychlení. Tentokrát ovšem není shora omezeno! Gustafsonův zákon v žádném případě Amdahlův zákon nepopírá, říká něco jiného. Amdahl řeší zrychlení pro problémy s konstantní velikostí a s konstantní sekvenční částí výpočtu. V tomto případě je zrychlení skutečně omezeno. Gustafson si za konstantu volí dobu běhu na paralelním systému a říká, že zrychlení není omezeno, ovšem za předpokladu dostatečně velkého problému. Gustafsonův zákon je matematicky vyjádřen ve vzorci 3.5.

Rozdíl mezi Amdahlovým a Gustafsonovým zákonem se dá jednoduše přiblížit na příkladu z oboru mechaniky. Amdahl říká, že pokud má auto ujet 60 km a během 1. hodiny urazilo 30 km, tak již na zbylých 30 km nikdy nemůže dosáhnout průměrné rychlosti 60 km/hod i kdyby jelo nekonečnou rychlostí. Gustafson oproti tomu říká, že pokud bude vzdálenost dostatečně velká (větší než 60 km), může se dosáhnout jakéhokoliv zrychlení. Zrychlení 90 km/hod dosáhne například tak, že pojedeme další hodinu rychlostí 150 km/hod. Důležité je si uvědomit, že oba zákony popisují něco jiného. Amdahlův zákon vychází ze sekvenčního výpočtu a odvozuje, kolikrát rychlejší může být tento výpočet s využitím paralelizace. Gustavsonův zákon vychází z paralelního výpočtu a vyvozuje, kolikrát déle by trval tento výpočet bez paralelizace.

Závěr pro problematiku prohledávání stavového prostoru je takový, že pro dostatečně velký stavový prostor s velkým počtem procesorů se zrychlení bude zvyšovat. Jinak řečeno, za stejný čas můžeme dosáhnout přesnějšího výsledku.

$$s = \frac{P_s(n)}{P_s(n) + \frac{P_p(n)}{p}} \tag{3.4}$$

$$S(n,p) \leq p + (1-p) * s \tag{3.5}$$

3.3 Synchronizace paralelního prohledávání

Jak už bylo zmíněno výše, paralelní algoritmy zahrnují i určitou režii, což snižuje jejich efektivnost. Existuje základní tři druhy režie (angl. *overhead*) [5]:

- Komunikační režie
- Synchronizační režie
- Režie při prohledávání

Komunikační režie se projeví například v paralelním zpracování algoritmu Alfa-beta, kdy oproti sekvěnčnímu zpracování musí procesor, který našel změnu pro hodnotu α nebo β, informovat i ostatní procesory.

Synchronizační režie se zase projeví v situaci, kdy například ve stavovém prostoru zpracovávají podstrom jednoho uzlu 4 procesory, každý jednu podvětev a 3 už prohledávání dokončili, nicméně musí čekat ve stavu *idle* na výsledky posledního procesoru.

Režii v prohledávání si můžeme demonstrovat také na příkladu algoritmu Alfa-beta, kde paralelní prohledávání může procházet uzly, které by se u sekvenčního prohledávání vůbec neprošly. To je způsobeno tím, že při začátku prohledávání není ještě známa hodnota alfa nebo beta z předchozích vedlejších větví.

Tyto jednotlivé režie jsou na sobě zdánlivě nezávislé. Pokud bychom chtěli některou zmenšit, může se to projevit na jiné režii. Když například snížíme prohledávací režii tak, že budeme častěji informovat ostatní procesory o mezivýsledcích prohledávání a tím zamezíme zbytečnému prohledávání některých uzlů, tak naopak zvýšíme komunikační režii.

3.4 Paralelní algoritmy pro prohledávání stavového prostoru

Je zřejmé, že ne každý uzel ve stavovém prostoru je stejně výhodný jako jiný. Zároveň ne na každém uzlu ve stavovém prostoru může dojít k α/β-ořezání. Dle podobných kritérií by bylo výhodné pojmenovat typy uzlů pro lepší orientaci v následujících algoritmech. Na Pan Donald E. Knuth a Ronald D. Moore v roce 1975 provedli základní rozdělení uzlů ve stavovém prostoru. Nějaké algoritmy jako například Principal Variation Splitting (PVSplit) z tohoto rozdělení vycházejí, některé jako například Young Brothers Wait Concept (YBWC) ho mírně upravují. Rozdělení uzlů je náledující:

- Typ 1: *PV uzly* (Principal Variation)
  Tvoří nejlevější větev seřazeného stromu (Principal Variation). Tyto uzly se musejí prohledat jako první, aby se výhodně nastavilo $\alpha\beta$ okénko. Paralelní prohledávání může začít až v momentě, kdy jsou tyto uzly prohledány.

- Typ 2: *CUT uzly*
  Přímí potomci PV uzlů a ALL uzlů. Potomci CUT uzlů mohou být ořezáni, proto se pokládá otázka, zda je moudré tyto potomky prohledávat paralelně.

- Typ 3: *ALL uzly*
  První přímý potomek CUT uzlu. Všechny potomky ALL uzlů musíme prohledat, proto je určitě vhodné tuto část paralelizovat.

3.4.1 Principal Variation Splitting (PVSplit)

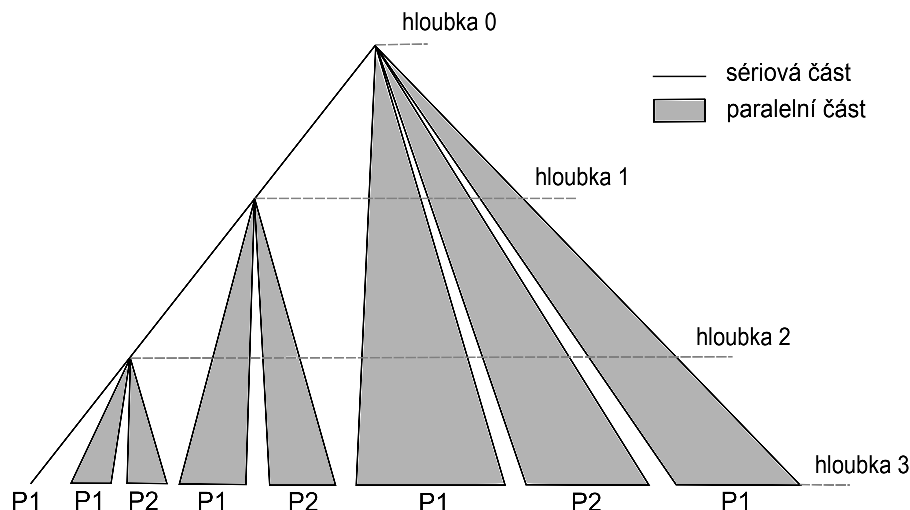
V tomto algoritmu musí být nejdříve sekvenčně prohledány PV uzly, aby se výhodně nastavilo $\alpha\beta$ okénko. Všechna vlákna nejdříve sestoupí na předposlední PV uzel. Odtud jeden procesor prohledá nejnižší PV uzel a ostatní mezi tím čekají. Až se vyhodnotí, tak si každý procesor vezme jednu další větev a vyhodnotí ji. V momentě, kdy procesor zpracuje danou větev, tak zpřístupní výsledky ostatním procesorům. Pokud nějaký procesor dokončí prohledávání a není tu žádná další ještě neprohledaná větev, tak čeká, než ostatní dokončí svoji činnost. V momentě, kdy jsou všechny větve aktuálního PV uzlu prohledány, vystoupí se na rodiče daného PV uzlu a takto se postupuje až ke kořenovému uzlu. Metoda PVSplit má relativně velkou režii při prohledávání, která se zejména projeví, pokud první větev není strategicky výhodná. Pokud algoritmus nalezne nejlepší hodnotu již v první větvi, tak se budou prohledávat ty samé uzly jako u sekvenčního zpracování, žádný navíc. Maximální zrychlení této paralelní metody bude jistě limitováno dle Amdahlova zákona. Princip je znázorněn také na obrázku číslo 3.1. Při 8 procesorech lze dosáhnout zrychlení až $4,1\times$ a třeba při 16 procesorech až $4,6\times$. Zrychlení by ale nemělo překročit hodnotu $5\times$. Zpoždění, způsobené sekvenčním prohledáváním a synchronizační režií, je totiž značné. [3]

Algoritmus 2: Principal Variation Splitting

```
if node is a terminal node or depth=maxDepth then
    return the heuristic value of node
endif
PVChild := PVParent.getFirstChild()
doMove(PVChild)
score:=-PVSplit(PVChild, depth+1, -beta, -alpha)
undoMove();
if score > alpha then
    alpha:=score
    if depth = 0 then
        setBestMove(PVChild)
    endif
endif
// PARALLEL PART: start
while PVParent.getNextChild() do
    doMove(child);
    score:=-NegaScout(child, depth+1, -beta, -alpha)
    undoMove()
    if score > alpha then
        alpha:=score
        if depth = 0 then
            setBestMove(child)
        endif
    endif
endw
// PARALLEL PART: end
return alpha
```

3.4.2 Young Brothers Wait Concept (YBWC)

Metoda dostala název podle toho, že se v každém uzlu nejdříve vyhodnotí první větev (*eldest brother*) a teprve potom ostatní větve (*young brothers*). YBWC definuje vztahy *Master/Slave* (neboli *klient/server*) mezi procesory. Když procesor vlastní uzel (*master*) je zodpovědný za jeho prozkoumání. Na začátku dostane jeden procesor vlastnictví nad root uzlem. Ostatní procesory jsou ve stavu *idle*. Tento procesor náhodně vybere jeden z volných procesorů a přiřadí mu větev od tzv. split pointu (uzel jehož hodnota první větve je již známa). Pokud je split pointů více, tak se přiděluje ten s nejmenší hloubkou ve stromu. *Slave* procesor má za úkol vyhodnotit danou větev a předat pak výsledky master procesoru. Procesor může narazit na podmínku α/β prořezání, potom se všechny slave procesory vracejí do stavu idle. Tato metoda je velice podobná PVSplit metodě, avšak YBWC může paralelně prohledávat v každém uzlu kromě prvního dítěte kořenového uzlu a ne jenom v PV uzlech jako je tomu u předchozí metody. Je zde méně synchronizační komunikace. Může být efektivně využito více procesorů. Je možné používat nesdílenou paměť. Zrychlení při pouužití 256 procesorů může být až 137×, což je výrazné urychlení oproti metodě PVSplit. [5]



Obrázek 3.1: Principal Variation Splitting (PVSplit).

3.4.3 Dynamic Tree Splitting (DTS)

Metoda používá *peer-to-peer* přístup oproti metodě YBWC, která používá přístup *master-slave*. Vlastnictví uzlu má odlišný význam v DTS než v YBWC. Několik procesorů může spolupracovat na expandování jednoho uzlu v daných větvích, ale za předání výsledku je odpovědný procesor, který má svůj dílčí výsledek jako poslední. U YBWC zodpovídal za vyřešení master (vlastník uzlu). Metoda má komplikovanu klasifikaci jednotlivých uzlů.

Na začátku je jeden procesor vybrán k prohledávání Root uzlu, zatímco ostatní uzly jsou ve stavu idle. Procesor ve stavu idle se dívá na globální seznam aktivních *split-point* uzlů, aby našel práci. Pokud je split-point nalezen, tak se přidá k ostatním procesorům pracujícím na tomto uzlu, uzel je sdílen. Pokud na seznamu není žádný split-point, tak vysílá pocesor hromadnou zprávu HELP všem procesorům. Procesor, který je zaneprázdněn, udělá kopii podstromu, kterou prohledává a uloží ji do sdíleného prostoru. Procesor ve stavu idle potom v tomto podstromu hledá split-point. Pokud je nalezen, tak se zkopíruje do seznamu split-uzlů, procesor tedy následně získá práci a může se připojit k ostatním procesorům. Pokud není split-point nalezen, tak po nějaké době procesor znovu zasílá hromadnou zprávu HELP. Díky tomuto principu může jeden procesor pomoci jinému v momentě, kdy prohledal svou podvětev a jiný ještě ne.

Tato metoda je využívána spoustou šachových programů. Metoda tráví trošku déle času při rozhodování, zda bude daný uzel split-point, tím vylepšuje prohledávácí parametry. Může používat distribuovanou (jednodušší implementace, pomalejší čas odpovědi) i sdílenou paměť (režie zamykání sdílené paměti, rychlejší čas odpovědi). Při použití 4 procesorů dosahuje metoda zrychlení $3,7\times$, při 8 procesorech $6,6\times$, při 16 procesorech až $11,1\times$. [5]

Kapitola 4

Realizace aplikace

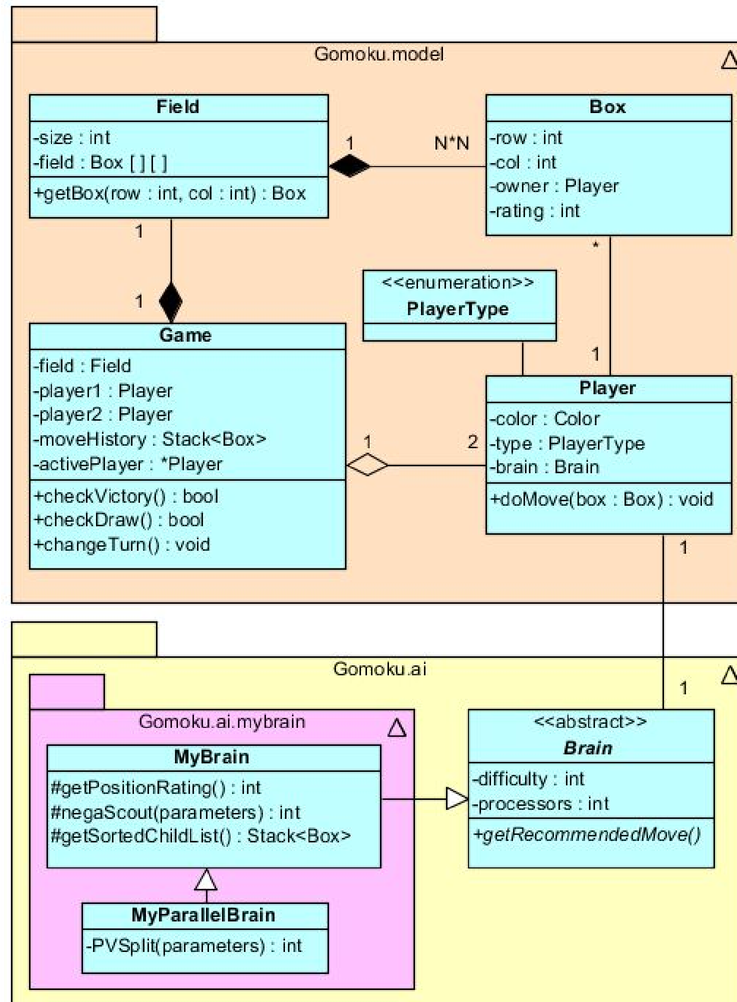
Aplikaci jsem se rozhodl naimplementovat v jazyce *Java*. Vybral jsem si tento jazyk, protože jsem chtěl vytvořit aplikaci postavenou na čistě objektovém návrhu. Další důvod, který mě vedl k tomu vybrat si tento jazyk, byla přenositelnost aplikace. Aplikace psaná v javě po kompilaci vytvoří spustitelný `jar` soubor, který se dá spustit napříč všemi známými operačními systémy, které mají nainstalován virtuální stroj jazyka Java. Jednoduchým rozšířením se dá následně aplikace vložit i na internet v podobě *appletu*. V neposlední řádě je Java jazyk, ve kterém mám již zkušenosti a ušetřil jsem tedy čas na samotné implementaci, který jsem pak mohl věnovat do vývoje umělé inteligence.

4.1 Návrh objektového modelu

Pro vytvoření aplikace jsem se rozhodl použít kompletně objektový návrh. Objektový návrh je vhodný, protože intuitivně reprezentuje objekty reálného světa. Implementované třídy disponují navenek pouze důležitými operacemi, vše ostatní je zapouzdřeno. Díky objektovému návrhu se stal celý projekt snadno upravovatelný a rozšiřitelný.

4.1.1 Třída Game

Základní třída `Game` zapouzdřuje celou logiku hry. Třída musí obsahovat atribut reprezentující informace o hrací ploše a dále musí obsahovat 2 atributy, reprezentující hráče. Třída `Game` musí umět uchovávat historii tahů, k tomu nám poslouží klasický zásobník, do kterého se budou ukládat jednotlivé tahy (reference na objekty třídy `Box`). Základní metody, které s instancí třídy `Game` musí pracovat je metoda testující konec hry. Konec hry nastává buď při výhře jednoho z hráčů, nebo při remíze. K remíze dojde tak, že se obsadí všechna pole, aniž by někdo vyhrál. K tomu, abychom mohli zjistit konec partie, budeme využívat veřejné metody třídy field (třída field reprezentuje hrací plochu). Bude výhodné, aby třída game obsahovala číselnou konstantu reprezentující počet kamenů nutných k výhře. Tuto konstantu pak budeme používat při určování výhry, při ohodnocování políček a všech dalších činností. Má to tu výhodu, že prostou změnou konstanty můžeme zkusit hrát hru s jiným počtem vítězných kamenů. Objekt game také bude muset vést hru. Když zahraje hráč tah, musí objekt game přepnout aktivitu na druhého hráče. K tomu nám poslouží reference na aktivního hráče, který bude objekt objekt game měnit při zahrání tahu.



Obrázek 4.1: Zjednodušený diagram tříd reprezentující základní strukturu projektu.

4.1.2 Třída Field

Třída `Field` bude reprezentovat hrací plochu. Základní atribut, který musí obsahovat je dvojrozměrná matice (naplněná objekty třídy `Box`), která reprezentuje čtvercovou hrací plochu. Velikost hrací plochy se vygeneruje podle atributu size, který musí být měnitelný, abychom mohli nastavovat různou velikost hrací plochy. Implicitně bude nastavena 15×15.

4.1.3 Třída Box

Třída `Box` reprezentuje konkrétní hrací pole. Musí obsahovat souřadnice na hracím poli a dále informaci o tom, zda ho už některý hráč obsadil. Třída `Box` bude dále obsahovat informaci o jeho aktuálním ohodnocení. To se bude samozřejmě každým tahem v závislosti na hráči, který je na tahu, měnit. Důležitým vylepšením budou reference na sousední boxy. Při testování výhry nebo při ohodnocování stavu hry se totiž prochází hrací pole ve všech směrech. Bylo by neefektivní neustále počítat indexy, na které se máme v matici přesunout. Proto je výhodné používat konstantní referemce na sousedy pro daný typ průchodu. Pokud

máme například průchod ve směru na východ, tak na levém kraji hrací plochy bude reference na východního souseda nastaven na nejpravější box na dalším řádku. Poslední box bude ukazovat na null hodnotu. Díky této optimalizaci se vše podstatně zrychlí. Další přídavný atribut bude reference na dočasný zásobník, ve kterém se daný box nachází. To se využije při paralelních průchodech stavového prostoru, kdy se do zásobníku budou ukládat klony původních boxů a každé vlákno bude moci pracovat nezávisle. Bude tedy potřeba ve třídě implementovat rozhraní `Cloneable`. Dále bude potřeba přetížit metodu `equals()` a metodu `hashCode()`

4.1.4 Třída Player

Třída `Player` bude reprezentovat hráče. Musí obsahovat typ hráče (HUMAN nebo PC), abychom mohli zadat hru jak dva počítače proti sobě, tak dva lidé proti sobě. Každý objekt třídy Player nese samozřejmě informaci o tom, zda vlastní černé kameny, nebo bílé. Důležitým atributem je objekt třídy `Brain`. Každý hráč totiž musí mít nastavenou nějakou podporující inteligenci. Počítač musí mít inteligenci, která mu bude doporučovat tah a hráčovi se hodí také pomocná inteligence. Z výukových důvodů totiž může chtít uživatel zjistit, který tah by mu doporučil počítač. Pak se již může rozhodnout jak chce. Každému hráči se tedy může nastavit jiný brain. Toto rozšíření nebude implementované, protože nehraje roli v zadání práce, ale z hlediska rozšiřitelnosti by nebyl problém to v tomto objektovém návrhu dodělat.

4.1.5 Třída Brain

Třída `Brain` je základní abstraktní třída, od které budou dědit všechny vytvořené umělé inteligence. Pěkně nám totiž implementovanou inteligenci zapouzdřuje pro jednotnou práci. Třída `Brain` musí obsahovat úroveň obtížnosti, kterou si již rozšiřující třída zpracuje po svém a další nutný atribut bude počet procesorů, který se má zapojit. Brainy, které nemají implementované paralelní algoritmy, tuto proměnnou používat nemusí. Základní metoda, kterou musí implementovat všechny brainy bude metoda `getRecommendedMove()`, která bude vracet referenci na konkrétní volný box.

Pro zajištění kompatibility brainů jsem se rozhodl použít spíše abstraktní třídu než rozhraní, protože abstraktní třída nám bude disponovat základními metodami, které by byly jinak duplicitní.

4.2 Návrh umělé inteligence

Dle analýzy možných algoritmů a postupů uvedených v kapitole 2 jsem si zvolil několik metod, které jsem použil v implementaci své umělé inteligence. Nejprve bylo potřeba navrhnout vhodnou ohodnocovací funkci. Následně jsem dodělal sériové prohledávání stavového prostoru. Nakonec jsem dané algoritmy zparalelizoval.

4.2.1 Ohodnocovací funkce stavu hry

K tomu, aby počítač mohl vyhodnotit daný stav hry, je potřeba mít naimplementovaný efektivní průchod maticí reprezentující hrací plochu. K tomu jsem využil referencí na sousedy boxů v daném směru prohledávání. Jednoduchou konstrukcí dokážu projít celé hrací pole. Při implementaci jsem pouze využil referenci na metodu, která vrací konkrétního souseda dle směru průchodu. Následující řádek pouze demonstruje zvolené jednoduché řešení.

```
while((nextBox = nextBox.getNeighbour()) != null) {...}
```

Pro vyhodnocování jsem se rozhodl využít vyhledávání vzorů na hrací ploše. Při konkrétním průchodu si efektivně uchovávám kombinaci posledních N kamenů ($N = 5$) v primitivním datovém typu integer. Integer bude tvořit klíč do hashovací tabulky vzorů, která obsahuje daná ohodnocení. Klíč se nejdříve vynuluje. Načteme další pole a vlastníka reprezentujeme číslem 0 (prázdné pole), 1 (vlastník je hráč, který je na tahů; MAX) nebo 2 (vlastník je druhý hráč; MIN). Identifikátor vlastníka se tedy vejde na pouhé 2 bity. Když zjistím id vlastníka, tak číslo přičtu ke klíči a udělám bitový posun o 2 bity doleva. Následně zamaskuji všechny bity vyšší než $(2 * N - 1)$ bitovou operací `and`. Tímto postupem dokážu extrémně rychle uchovávat kombinaci posledních N kamenů v řadě. Následně se algoritmus podívá do hashovací tabulky (velice rychlý průchod) a pokud v ní vzor najdu, tak přičtu jeho hodnotu k celkovému ohodnocení stavu. Některé vzory mají kladné ohodnocení (výhodné pro hráče na tahu), jiné zase záporné. Výsledné číslo ohodnocení tedy reprezentuje výhodnost stavu hry pro hráče, který je na tahu. Pokud naleznu vítěznou kombinaci, tak vracím její příslušné maximální možné ohodnocení a prohledávání ukončím.

V tabulce 4.1 je uvedený seznam vzorů, který se hodnotí, doplněný o příslušné ohodnocení. Sloupec *vzor* obsahuje pěticiferné číslo představující pětici políček v řadě na hrací ploše. Hodnota každé cifry představuje obsazenost políčka se stejným významem jako bylo popsáno v předchozím odstavci. Tímto způsobem jsou nadefinovány všechny vzory. Číslo reprezentující tvar vzoru je v samotném zdrojovém kódu ve formě řetězce, který se v konstruktoru třídy Brain převede na číslo v intervalu $\{0 \dots 682\}$. Nejvyšší vzor může být 22222 dekadicky, což se dá po cifrách vyjádřit jako binární číslo ve tvaru 1010101010. To již odpovídá výše zmíněné hodnotě 682. Následující transformace klíče má význam z hlediska optimalizace rychlosti vyhledávání v hashovací tabulce. Vše se děje pouze při inicializaci objektu Brain, případně při změně obtížnosti. Na daný výběr vzorů a příslušné ohodnocení jsem přišel empirickým způsobem. Přidáním dalších vzorů a změnou jejich ohodnocení se může výrazně změnit styl hry. Funguje to ale jako dvousečná zbraň. Pokud se zvýhodní například ohodnocení dvojic a trojic a hráč začne hrát tedy více pozičně, nemusí pak rozpoznat výhodný tah vedoucí k jasné výhře, který vede před vynucený tah. Proto je potřeba měnit ohodnocení s rozvahou a vše řádně testovat.

Styl hry se dá upravit vynásobením konstant další rozšiřující konstantou. Tím lze vlastně upravovat poměr útoku vůči obraně. V této implementaci je například použit koeficient, který upravuje základní ohodnocení vzorů v závislosti na tom, zda je hloubka prohledávání

lichá nebo sudá a v závislosti na tom, zda vlastní hráč černé nebo bílé kameny. Představme si situaci, že má bílý hráč vytvořenou nehlídanou trojici kamenů a černý hráč pouze dvojici. Při hloubce prohledávání 3 může nastat situace, že by logika uvažovala tak, že nejdříve černý hráč vytvoří ze své dvojice trojici, následně bílý hráč z trojice čtveřici a nakonec černý hráč z trojice také čtveřici. Pak mají v hloubce 3 oba hráči jednu čtveřici, což by se dalo interpretovat jako vyrovnaná situace, ale přitom bílý hráč dalším tahem vyhrává. Proto je dobré koeficientem upravit sílu vzorů. Čím je ale nastavena vyšší lichá hloubka, tím více se tento problém smazává. Další ovlivnění vzorů mám nastavené v závislosti na tom, zda má hráč černé nebo bílé kameny. Dle mého názoru by začínající hrač měl využít svého výsadního postavení a měl by spíše utočit, kdežto bílý hráč bude nucen z toho samého důvodu zase spíše bránit. Během určitého počtu tahů se tato situace sice změní, ale další heuristika, která by vlastně měnila poměr útoku a obrany během hry by mohla zpomalit celý vyhodnocovací proces a proto jsem tuto heuristiku dále nerozvíjel.

Vzor	**Ohodnocení**	**Vzor**	**Ohodnocení**
11111	MAX	22222	MIN
01111	220	02222	−220
11110	220	22220	−220
11011	200	22022	−200
10111	200	20222	−200
11101	200	22202	−200
01110	120	02220	−120
01100	20	02200	−20
00110	20	00220	−20
01010	20	02020	−20

Tabulka 4.1: Tabulka vyhledávaných vzorů a jejich ohodnocení

4.2.2 Prohledávání stavového prostoru

K průchodu stavového prostoru jsem použil algoritmus *NegaScout* (1), který je z algoritmů uvedených v sekci 2.2 nejlepší.Abych mohl ovšem tento algoritmus použít, musel jsem naimplementovat vhodné řazení tahů.

Řazení provádím speciální metodou, která mi vrací potomky prohledávaného uzlu ve správném pořadí. Metoda ohodnocuje potomky na základě toho, zda jsou součástí nějakého vzoru. (Vzory jsou již nadefinované pro ohodnocovací funkci.) Ohodnocování funguje prakticky na stejném principu jako v ohodnocovací funkci, ale prohledávají se jen sousedi boxu do vzdálenosti N. Pokud je vzor součástí nějakého vzoru, nastavíme boxu patřičné ohodnocení a přidáme ho do seznamu. Abychom nepřišli o nějaký dobrý tah, který nezapadá do žádného z vybraných ohodnocených vzorů, tak mám nadefinovo několik dalších vzorů, které sice mají nulové ohodnocení, ale mohli by být výhodné. Všechny nadefinované vzory jsou potenciálně rozšiřitelné na vítěznou kombinaci. Nikdy se tedy nebude prohledávat pozice, která už nemůže být součástí výherní kombinace. Tímto je naimplementovaná tzv. *detekce mrtvých tahů.* Zbytek pozic, které se neprohledávají by neměly být výhodné, protože vycházím z předpokladu, že každá dobrá situace se skládá ze sekvence alespoň trochu dobrých. Nepředvídatelný tah u hry jako je gomoku může způsobit okamžitou prohru

hráče, který tah udělal (rozdíl oproti složitější hře šachy). Velká převaha se dá totiž získat velice rychle a snadno i pouze jedním špatným tahem. Pro seřazení seznamu využívám standartní metody `Collections.sort(list)`, která používá speciálně vytvořený komparátor třídy `BoxComparator`, která řadí boxy na základě předchozího ohodnocení.

4.2.3 Paralelizace algoritmů

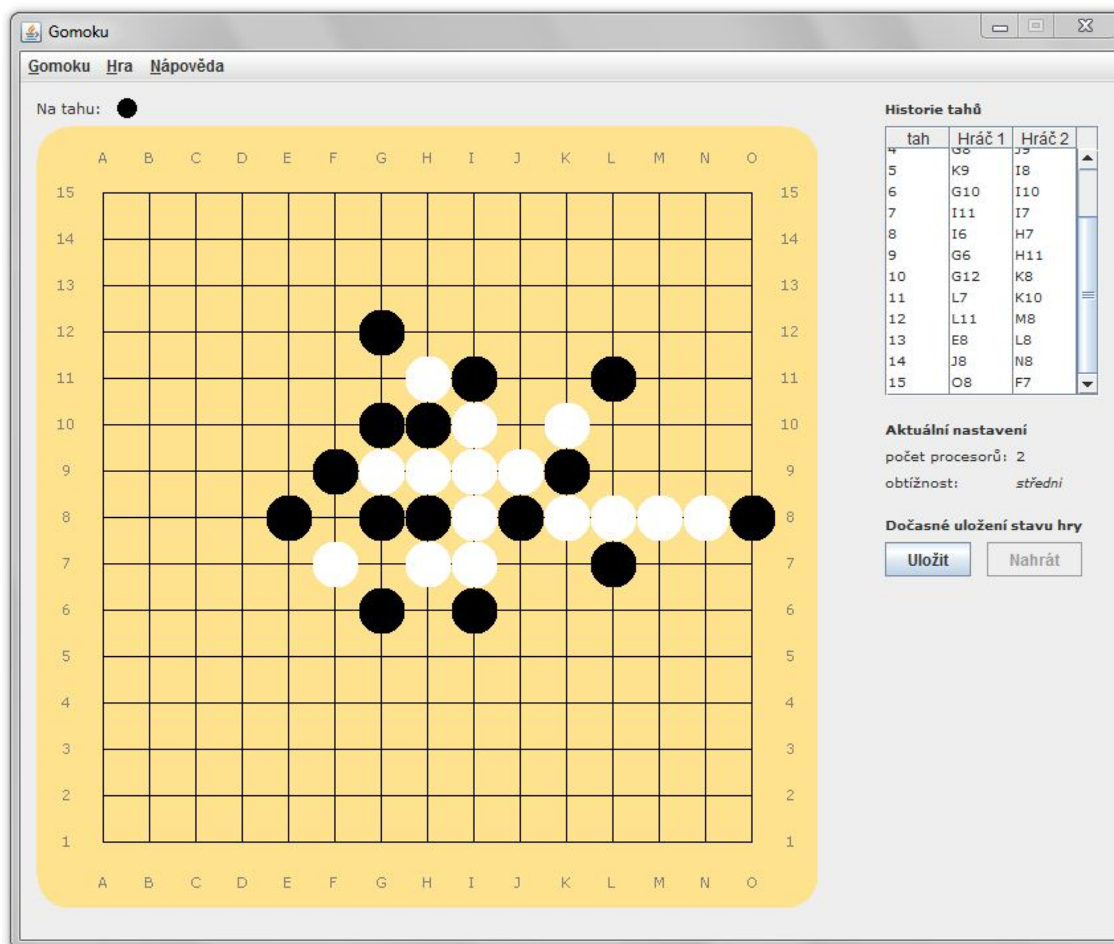
K samotné implementaci paralelismu jsem využil klasických nativních vláken v javě. Spuštěné vlákno vždy vykonává obsah metody `run()` ve třídě implementující rozhraní `Runnable`. Tato metoda obsahuje právě paralelní část algoritmu *PVSplit*. Vzhledem k tomu, že jednotlivá vlákna musejí v PV uzlech sdílet proměnné alpha, beta a zásobník se zbývajícími potomky PV uzlu, musel jsem práci s těmito proměnnými synchronizovat. Atomickou úpravu těchto proměnných zajistí speciální `synchronized` bloky, která dokážou dát proměnné zámek, takže ji může modifikovat zároveň pouze jedno vlákno.

Další problém, který jsem musel vyřešit bylo zajištění, aby se hodnoty alpha a beta přenášely do rutin vláken odkazem. Proto jsem nemohl použít ani primitivní datový typ `int`, ani jeho objektovou reprezentaci `Integer`. `Integer` se totiž v Javě při editaci nemění, ale znovu vytváří. Je to nezměnitelná (*immutable*) proměnná. Proto jsem musel využít alternativního datového typu `MutableInt`, který datový typ `int` objektově zapouzdřuje. Další řešení by bylo nastavit proměnné jako atributy třídy `Brain`, ale to jistě není optimální řešení.

Synchronizace editace proměnných je tedy vyřešena, musela se ale také zajistit synchronizace vláken. V momentě, kdy je prázdný zásobník s potomky PV uzlu, tak se musí počkat, než všechna vlákna dokončí svoji rutinu. K tomu se dá využít standardní metody `join()` nad vláknem. Tato metoda sváže aktuální vlákno s vybraným vláken. Aktuální vlákno se uspí a probudí se až v momentě, kdy vybrané vlákno dokončí svoji činnost.

4.3 Grafické rozhraní

Grafické rozhraní jsem vytvořil v grafickém výjovém prostředí *NetBeans* (IDE 7.0 Beta2). Rozhraní využívá stadardní grafické knihovny *Swing* a *AWT*. Zadání práce ovšem kladlo důraz spíše na umělou inteligenci než na vzhled, proto bylo grafické rozhraní vytvořeno tak, aby vyhovovalo hlavně z funkční stránky. Na obrázku 4.2 je znázorněn vzhled vytvořené aplikace.



Obrázek 4.2: Grafický vzhled aplikace.

4.3.1 Ovládání aplikace

Okno aplikace se skládá z hrací plochy, bočního přehledového sloupce a menu. Po spuštění aplikace automaticky začíná hra. Implicitně je uživatel na tahu a hned může zahrát tah kliknutím na hrací plochu. Nyní se pokusím vysvětlit základní ovládání aplikace.

Novou hru spustíte kliknutím na tlačítko *Hra/Nová* nebo klávesovou zkratkou `Ctrl+N`. Pokud by chtěl uživatel hrát za hráče s bílými kameny nebo chtěl třeba jen nastavit, aby mohl hrát s jiným uživatelem duel, tak najde nastavení v menu Hra, kde může nadefinovat typ obou hráčů (člověk nebo PC).

Obtížnost a počet procesorů pro paralelní výpočet se dá nastavit v menu Hra. Předpokládá se, že nejtěžší obtížnost hry budete nastavovat v kombinaci s více zapojenými procesory. V opačném případě bude výpočet velice pomalý. Aktuální nastavení obtížnosti a počtu zapojených procesorů také vidíte v bočním přehledovém sloupci. Obtížnost a počet procesorů se dají měnit i pomocí příslušných klávesových zkratek. Nastavení se může měnit i během hry a nabývá platnost od následujícího tahu.

V bočním přehledovém menu je k dispozici panel s historií tahů. Pokud by měl uživatel zájem si zkusit dočasně uložit nějaký stav hry a vyzkoušet více variant tahům které ho napadnou, může použít funkce dočasného uložení hry a hru následně kdykoliv zase nahrát. Ukládání stavu hry je dočasné, proto je po ukončení programu zahozeno. Gomoku je hra, která se většinou hraje jen pár minut, proto nemá moc velký smysl partii nějak externě ukládat.

Velice snadno si lze zrekonstruovat libovolnou partii. Nastaví se hra člověk$\times$člověk, vytvoří se požadovaná herní pozice a nakonec se jen u jednoho z hráčů změní typ na PC.

Hra se dá ukončit křížkem v pravém horním okraji okna, tlačítkem *Gomoku/Konec* nebo klávesou zkratkou `Ctrl+Q`. Rozšiřující informace o programu naleznete v okně, které se otevře stisknutím tlačítka *Nápověda/O programu*.

Kapitola 5

Dosažené výsledky

V této kapitole bych chtěl shrnout výsledky mé práce. Nejdříve se zaměřím na zhodnocení konkurenčních programů hry gomoku (piškvorky) a následně zhodnotím přínos paralelizace algoritmů.

5.1 Testování síly umělé inteligence

Pro otestování síly vytvořené umělé inteligence jsem použil dvě volně dostupné implementace hry gomoku a dále jsem nechal hrát program proti hráči s vysokým rankingem z prestižního serveru *playok.com*. Testování proti jiným programům jsem prováděl tak, že jsem si otevřel jak můj, tak cizí program a hrál jsem paralelně obě partie. Tah počítače v jedné hře jsem nastavil jako lidský tah ve druhé hře a zase naopak. Testování je nutné provést i s člověkem, protože živá složka představuje vysokou míru přirozené inteligence a kreativity. Člověk může měnit styl hry jen díky pouhé intuici, kdežto počítač musí mít veškeré své chování přesně nadefinované. Může se tedy stát, že chování počítače bude předvídatelné. Tomu by se ovšem mělo zabránit určitou mírou nedeterministického rozhodování.

5.1.1 Pisky 1.3 (konkurenční aplikace)

Program Pisky byl první program, který jsem testoval proti mému programu Gomoku. Samotná stažená verze má označení 1.3 i přesto, že je na webu uvedeno, že se jedná o verzi 1.0. V této aplikaci se nedá nastavit konkrétní obížnost hry, ale je tu nastavitelný procentuální poměr útoku hry protihráče vůči jeho obraně. Standartně je nastevn útok 60% a obrana 40%. Testování jsem navíc provedl pro herní styl 50% – 50% a 40% – 60%. Nyní uvedu několik záznamů z partií s tímto programem. Aplikace Pisky se dá volně stáhnout na webové adrese `http://www.slunecnice.cz/sw/gomoku/`.

- Gomoku (*obtížnost těžká*) vs. Pisky 1.3 (*60% útok, 40% obrana*)
  Moje aplikace má **černé** kameny: prohra

  ```
  1.I9, 2.I8, 3.J8, 4.K7, 5.K9, 6.J9, 7.K10, 8.L10, 9.K11, 10.K12, 11.L12, 12.J10, 13.J11,
  14. L11, 15.L9, 16.I12, 17.M8, 18.N7, 19.N9, 20.L7, 21.M9, 22.O9, 23.M7, 24.M6, 25.K8,
  26.M10, 27.L8, 28.N8, 29.K13, 30.J13, 31.I14, 32.H11, 33.K14, 34.J14, 35.J12, 36.I11,
  37.N6, 38.O5, 39.J7, 40.I6, 41.O15, 42.K4, 43.L5, 44J5, 45.N14, 46.M13, 47.H7, 48.G10,
  49.F9, 50.F10, 51.H10, 52.G11, 53.E9, 54.G9, 55.G12, 56.E11, 57.D14, 58.F11
  ```

  Moje aplikace má **bílé** kameny: remíza

- Gomoku (*obtížnost těžká*) vs. Pisky 1.3 (*50% útok, 50% obrana*)
  Moje aplikace má **černé** kameny: vítězství

  `1.J10, 2.J11, 3.K11, 4.I9, 5.K9, 6.K10, 7.L9, 8.L8, 9.M9, 10.N9, 11.N8, 12.L10,13.M10, 14.J9, 15.M8, 16.L11, 17.M7, 18.M11, 19.M6`

  Moje aplikace má **bílé** kameny: vítězství

  `1.H8, 2.G7, 3.G8, 4.F8, 5.H6, 6.H7, 7.F7, 8.H9, 9.G9, 10.I7, 11.J7, 12.I6, 13.I8, 14.K8, 15.J9, 16.J6, 17.K7, 18.L6, 19.M6, 20.L5, 21.L4, 22.M5, 23.K5, 24.M4, 25.K4, 26.K3, 27.I4, 28.J4, 29.H5, 30.G6, 31.G5, 32.I5, 33.L2, 34.N3, 35.L3, 36.O2, 37.K6, 38.J5, 39.J3, 40.L7, 41.L8, 42.M3, 43.K2, 44.L1, 45.J2, 46.M2, 47.M1, 48.H2, 49.H4, 50.I3, 51.E4, 52.N4, 53.F4, 54.G4, 55.E3, 56.D2, 57.E5, 58.E6, 59.F5, 60.D5, 61.D4, 62.F2, 63.D3, 64.E2, 65.F3, 66.G2`

- Gomoku (*obtížnost těžká*) vs. Pisky 1.3 (*60% útok, 40% obrana*)
  Moje aplikace má **černé** kameny: vítězství

  `1.H8, 2.G8, 3.G9, 4.F10, 5.H10, 6.H9, 7.I10, 8.J11, 9.I9, 10.J10, 11.I11, 12.I12, 13.J12, 14.F8, 15.K13`

  Moje aplikace má **bílé** kameny: vítězství

  `1.H8, 2.I7, 3.H7, 4.H9, 5.I8, 6.G8, 7.I10, 8.G6, 9.G7, 10.I9, 11.J9, 12.J8, 13.H9, 14.H10, 15.G11, 16.K7, 17.L7, 18.L6`

Z následujícího záznamu her je vidět, že tato konkurenční aplikace je zřejmě slabší než moje vytvořená aplikace. Pokud kladla aplikace *Pisky* větší důraz na útok, tak dokázala mé aplikaci konkurovat i zvítězit, ale ve vyrovnaném důrazu na útok i obranu nebo při důrazu na obranu se konkurence neudržela a výrazně zaostávala.

Při testování aplikace *Pisky* s mojí aplikací při střední obtížnosti byla herní síla relativně vyrovnaná. Jedna hra byla neplatná, protože aplikace *Pisky* nepodporuje rozšíření hry gomoku, kdy vyhrává právě 5 kamenů a ne více.

5.1.2 Piškvorky 8.4 (konkurenční aplikace)

Další program, se kterým jsem moje Gomoku testoval byly *Piškvorky 8.4*. Autorem je Petr Laštovička a program vyvýjel v letech 2000 – 2009. Umělá inteligence těchto piškvorek je o několik tříd výše než u programu *Pisky 1.3*. S touto aplikací jsem provedl několik testování, ale bohužel zvítězila skrz všechny obtížnosti. Autor zřejmě za několik let vývoje velice vyladil heuristická ohodnocování stavu hry a tím dosáhl lepších výsledků i pro menší hloubku prohledávání. Navíc ve svém programu autor používá databázi počátečních tahů z externího souboru, což může hrát velkou roli, pokud si nechal tyto tahy zhodnotit nějakým profesionálním hráčem. Tato aplikace má lepší herní výsledky než moje, ale i tak byly partie do poslední chvíle vždy alespoň na první pohled vyrovnané. Nyní uvedu několik záznamů.

- Gomoku (*obtížnost těžká*) vs. Piškvorky 8.4 (*obtížnot 50.0 = nejvyšší*)
  Moje aplikace má **černé** kameny: prohra

1.H6, 2.G5, 3.I5, 4.G7, 5.H4, 6.J6, 7.I4, 8.G4, 9.G6, 10.H5, 11.I6, 12.I7, 13.I2, 14.I3, 15.H7, 16.G8, 17.F6, 18.E6, 19.F5, 20.E4, 21.I8, 22.J9, 23.H8, 24.H9, 25.F7, 26.F8, 27.K4, 28.E7, 29.J5, 30.L3, 31.J4, 32.L4, 33.F3, 34.F4, 35.L2, 36.K3, 37.G2, 38.E5, 39.D15, 40.E3

Moje aplikace má **bílé** kameny: prohra

1.G7, 2.F8, 3.G6, 4.G8, 5.H8, 6.F6, 7.F9, 8.F7, 9.F4, 10.H9, 11.E6, 12.I10, 13.J11, 14.G5, 15.K11, 16.I11, 17.H10, 18.I9, 19.I8, 20.I12, 21.I13, 22.J9, 23.K9, 24.K8, 25.L7, 26.H11, 27.G12, 28.J13, 29.K14, 30.E7, 31.D8, 32.H4, 33.I3, 34.D6, 35.C5, 36.J12, 37.C6, 38.C7, 39.D7, 40.E8, 41.F5, 42.G4, 43.E4, 44.H7, 45.D3, 46.C2, 47.D5, 48.F3, 49.B9, 49.C8, 51.B5, 52.E5, 53.C4, 54.B8, 55.A9, 56.E2, 57.B7, 58.A8, 59.B3, 60.K3, 61.A2

Průměrná doba přemýšlení programu *Piškvorky 8.4* byla 3.730 sec a průměrná doba přemýšlení mého programu byla 0.605 sec.

- Gomoku (*obtížnost těžká*) vs. Piškvorky 8.4 (*obtížnot 15.0 = střední*)
  Moje aplikace má **černé** kameny: prohra

  1.I9, 2.H8, 3.J8, 4.H10, 5.I7, 6.K9, 7.J7, 8.H7, 9.H9, 10.I8, 11.J9, 12.J10, 13.J5, 14.J6, 15.I10, 15.K8, 17.G9, 18.F9, 19.G8, 20.F7, 21.J11, 22.K12, 23.K7, 24.F8, 25.L6, 26.M5, 27.M7, 28.L7, 29.F6, 30.H12, 31.I11, 32.K11, 33.K10, 34.I13, 35.G11, 36.G10, 37.K5, 38.J4, 39.G5, 40.F4, 41.G6, 41.G7, 43.I5, 44.H5, 45.N8, 46.O9, 47.H6, 48.D7, 49.O7, 50.E7

  Moje aplikace má **bílé** kameny: prohra

  1.G6, 2.F5, 3.G7, 4.G5, 5.F4, 6.E5, 7.D5, 8.H4, 9.I5, 10.H6, 11.H3, 12.I3, 13.J2, 14.E7, 15.F6, 16.E6, 17.E8, 18.E4, 19.E3, 20.H7, 21.H8, 22.D7, 23.C8, 24.F8, 25.G9, 26.F10, 27.I9, 28.J10, 29.F7, 30.D9, 31.H9, 32.J9, 33.F9, 34.E9, 35.G10, 36.G8, 37.F11, 38.I8, 39.G11, 40.K10, 41.L11, 42.H11, 43.G12, 44.G13, 45.F12, 46.J8, 47.E12, 48.D12, 49.D13

  Průměrná doba přemýšlení programu *Piškvorky 8.4* byla 2.3 sec a průměrná doba přemýšlení mého programu byla 0.6 sec.

5.1.3 Testování hry s profesionálním hráčem

Nakonec jsem nechal hru otestostovat téměř profesionálním hráčem. Pavel Šikula hraje na prestižním serveru *playok.com* a aktuální ranking má 1811. V České piškvorkové lize je po posledním kole dokonce v první dvacítce! Požadavek na něj byl takový, aby zkusil hrát nejprve partii s neomezeně dlouhým časem a následně aby se snažil, aby se jeho celkový čas na přemýšlení vešel do 1 minuty. V prvním testování se zkušenosti hráče projevily a vyhrál 2 : 0. V rychlém zápase skončila partie remízou 1 : 1. Nyní uvedu záznamy dvou her rychlé partie.

- Gomoku (*obtížnost těžká*) vs. Pavel Šikula (*ranking 1811 na serveru playok.com, „rychlovka“*)
  Moje aplikace má **černé** kameny: výhra

  1.I8, 2.I9, 3.J9, 4.H7, 5.J7, 6.J8, 7.K7, 8.L7, 9.K6, 10.L5, 11.J6, 12.K5, 13.L8, 14.I5, 15.M9, 16.N10, 17.H9, 18.G10, 19.J5, 20.J4, 21.K8, 22.K9, 23.I6, 24.H6, 25.L9, 26.G7, 27.M10

Moje aplikace má **bílé** kameny: prohra

```
1.H7, 2.I6, 3.J7, 4.K7, 5.H5, 6.H8, 7.J5, 8.I5, 9.I4, 10.J6, 11.H4, 12.L8, 13.M9, 14.K6,
15.H6, 16.H3, 17.K5, 18.L6, 19.M6, 20.J8, 21.I9, 22.M5, 23.N4, 24.I8, 25.K8, 26.G8, 27.F8,
28.L7, 29.L9, 30.L4, 31.L5, 32.K9, 33.J10, 34.M11, 35.L10, 36.I10, 37.F7, 38.J3, 39.F6,
40.F5, 41.G7, 42.I7, 43.E7, 44.M12, 45.D7
```

5.2 Výsledky paralelizace umělé inteligence

Zjištění skutečného zrychlení paralelního výpočtu nebylo jednoduché zjistit. Problém byl sehnat nějaký PC nebo server, který má dostatečně velký počet procesorů (jader). Testování jsem nakonec provedl na fakultním serveru `merlin.fit.vutbr.cz`, který běží na dvou čtyřjádrových Opteronech. Jediný problém zde byl takový, že server má nastavené omezení paměti (200 MB fyzické, 400MB virtuální) a limit pro přidělený procesorový čas. Nicméně to byl asi nejlepší stroj pro testování, který jsme spolu s mým vedoucím bakalářské práce sehnali. Zrychlení výpočtu jsem tedy zjišťoval pro $1-8$ paralelních větví. Testování jsem prováděl na různých herních situacích. Nejdříve jsem si uložil danou situaci v prostředí aplikace a následně jsem měřil čas reakce pro různý počet paralelních větví a různou hloubku prohledávání. Nakonec jsem hodnoty zpracoval.

Vzhledem k tomu, že algoritmus Principal Variation Search se dělí o prohledávání jen v PV uzlech a procesor vždy prozkoumává přidělou větev celou, napadla mne otázka, zda se zjištěná průměrná zrychlení trochu neliší pro různé hloubky prohledávání. Zjištěné výsledky zrychlení pro různé hloubky prohledávání demonstruje graf 5.1. Než jsem změřil výsledky, tak mě napadly dva faktory, které by mohly způsobovat rozdíly:

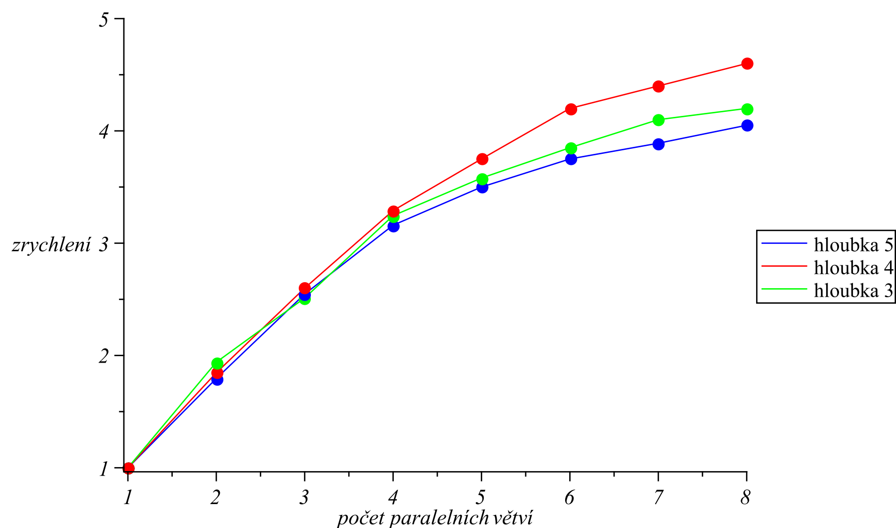
- Algoritmus Principal Variation Splitting si dělí práci pouze v PV uzlech, proto se občas může při větších hloubkách prohledávání neefektivně čekat na výsledek.
- Větší hloubka prohledávání přináší větší počet uzlů, které se budou prohledávat paralelně, kdežto počet PV uzlů se zvětší jen o příslušný rozdíl hloubky.

První rozdíl by spíše nahrával většímu zrychlení u menších hloubek prohledávání (u tohoto konrétního algoritmu), druhý rozdíl zase spíše nahrává větším hloubkám. Výsledky ovšem nepotvrdily ani jednu tezi. Testování probělo na situacích uvedených v příloze dokumentu v podkapitole B.

Vyzkoušel jsem také zjistit celkové zrychlení, které jsem vypočítal jako aritmetický průměr z hodnot příslušných pro dané hloubky prohledávání. Nenašel jsem totiž žádné jiné výsledky, které by zrychlení dělily na příslušné hloubky prohledávání. V získaném formátu jsem mohl zrychlení porovnat s konkurenčními výsledky. Zjištěné výsledky jsou uvedeny v tabulce 5.1.

Podobné výsledky uvadí ve své diplomové práci z roku 2001 pro Univerzitu v Torintu i Valavan Manohararajah [5]. Uvedené výsledky zrychlení má nepatrně menší. Pro 4 paralelních větví uvádí zjištěné zrychlení $3,0$, pro 8 větví $4,1$ a pro 16 větví $4,6$. Ve své práci píše, že se mu zdá, že zrychlení bude konvergovat k hodnotě 5. Tento úsudek mu bohužel potvrdit nemohu, protože jsem neměl k dispozici dostatečnou výpočetní sílu. Výsledky zrychlení se ale pro 6 a více paralelních větví začínají značně zmenšovat, což by zmíněné tezi nahrávalo. K mírně odlišným výsledkům mohlo dojít jinou interpretací pseudokódu a odlišnostmi v použitém programovacím jazyce. Řádově jsou ale výsledky stejné. Pseudokódů

k této problematice je velice málo a většinou jsou velice abstraktní, zjednodušené a často jen slovně vysvětlené.



Obrázek 5.1: Graf demonstruje jak se mění zrychlení pro různý počet paralelních větví. Pro porovnání jsou zvlášť uvedeny výsledky pro hlobku prohledávání 5 a 4 (nejvyšší měřené hloubky).

počet paralelních větví	1	2	3	4	5	6	7	8
zrychlení	1×	1,87×	2,55×	3,23×	3,61×	3.93×	4.13×	4.28×

Tabulka 5.1: Tabulka uvádí průměrné zrychlení pro příslušné počty paralelních větví u implementovaného algoritmu *Principal Variation Splitting*.

Kapitola 6

Závěr

Tato bakalářská práce měla za cíl vytvořit hru gomoku s inteligentním soupeřem, který prohledává stavový prostor paralelně. Následně se měl měl uvést přínos práce a porovnání s jinými implementacemi hry gomoku.

Při tvorbě umělé inteligence jsem řešil dilema, zda preferovat složitější ohodnocovací funkci stavu hry nebo zda raději prozkoumávat stavový prostor do větší hloubky. Pokud člověk nevlastní superpočítač, tak musí s přidělenými prostředky velice šetřit. Tento problém jsem vyřešil následovně. Ohodnocovací funkci jsem se snažil udělat co možná nejrychlejší. Proto dokáže pouze jedním průchodem matice ve 4 zakladních směrech ohodnotit stav hry. Prohledávání stavového prostoru jsem zjednodušil tak, že jsem využil algoritmy Negascout a paralelní algoritmus Principal Variation Splitting, které prohledávají stavový prostor v seřazeném pořadí od co možná nejlepšího potomka (v Principal Variation) po nejhoršího. Při mělkém ohodnocení stavu hry jsem byl schopen ohodnotit sílu jednotlivých tahů na základě jejich příslušnosti k ohodnoceným vzorům a filtrovat tak *mrtvé* tahy (nemohou být součástí výherní kombinace) a z největší pravděpodobnosti špatné tahy (tahy, které jsou vzdálené od dějiště hry). Touto cestou jsem dané dilema vyřešil a aplikace si zachovala očekávanou rychlost výpočtu.

Aplikace je napsána v plně objektovém modelu, který je snadno rozšiřitelný a udržovatelný. Docela jednoduše by se dal projekt upravit, aby podporoval turnaje ve hře gomoku nebo aby fungoval po síti. Při testování se mi podařilo uhrát solidní výsledky proti konkurenčním implementacím hry gomoku i proti kvalitnímu hráči z prestižního serveru *playok.com* (viz. podkapitola 5.1). Aby mohla ale aplikace konkurovat těm nejlepším, stálo by to ještě určitě mnoho úsilí. Možná by se třeba došlo k závěru, že bude vhodnější použít jinou logiku. Jedno z možných rozšíření by bylo doplnit aplikaci o databázi počátečních tahů. Počáteční tahy jsou velice důležité a hrají obrovský vliv na samotnou pratii. Tyto tahy by se musely prokonzultovat s profesionálními hráči, protože určit si tahy na základě samostatného testování by bylo více než troufalé. Zde má moje aplikace asi největší slabinu.

Myslím, že se mi podařilo splnit požadavky na aplikaci. Během řešení jsem nahlédl do principů tvorby inteligentního soupeře a i přesto, že hra gomoku patří jistě mezi jedodušší hry, tak principy na ní uplatněné by se zcela jistě daly použít i na složitější hru šachy nebo kteroukoliv jinou stavovou hru se dvěma soupeři. Zjistil jsem, že obor umělé inteligence je opravdu rozsáhlý a dostal jsem příležitost se jí i dále věnovat.

Literatura

[1] AMDAHL, G.: Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS '67 (Spring) Proceedings of the April 18-20*, New York: ACM, 1967, s. 483–485.

[2] DVOŘÁK, V.: *Architektury a programování paralelních systémů*. Brno: VUTIUM, 2004, ISBN 80-214-2608-X.

[3] GRESKAMP, B.: Parallelizing a Simple Chess Program. 2003, [online].
URL `http://iacoma.cs.uiuc.edu/~greskamp/pdfs/412.pdf`

[4] Jez9999: Alpha-Beta pruning example. [online], 2007-05-28 [cit. 2010-12-28].
URL `http://cs.wikipedia.org/wiki/Soubor:AB_pruning.svg`

[5] MANOHARARAJAH, V.: *Parallel Alpha-Beta Search on Shared Memory Multiprocessors*. Diplomová práce, University of Toronto, 2001, [online].
URL `http://www.valavan.net/mthesis.pdf`

[6] MASAŘÍK, V.; ŠTĚPÁNKOVÁ, O.; LAŽÁNSKÝ, J.; aj.: *Umělá inteligence (1)*. Praha: ACADEMIA, 2004, ISBN 80-200-0496-3.

[7] Němec, J.: Složitost alfabeta metody. [online], 2006-07-17 [cit. 2011-03-18].
URL `http://www.linuxsoft.cz/article.php?id_article=1239`

[8] PĚCHOUČEK, M.: Hraní dvouhráčových her, adversariální prohledávání stavového prostoru. [online], 2007-11-22 [cit. 2011-03-10].
URL `http://labe.felk.cvut.cz/~pechouc/kui/games.pdf`

[9] Česká federace piškvorek a renju: Gomoku official rules. [online], 2008-02-02 [cit. 2010-12-27].
URL `http://www.piskvorky.cz/en/federation/gomoku-official-rules/`

[10] RYBKA, A.: Historie gomoku a renju. [online], 2008 [cit. 2010-12-27].
URL `http://www.piskvorky.cz/clanky/zajimavosti-ze-sveta-piskvorek-a-renju/historie-gomoku-a-renju/`

Dodatek A

Obsah DVD

A.1 Spustitelný program

Spustitelná aplikace je uložena ve formátu `JAR`. Je spustitelná na všech operačních systémech, které mají nainstalovaný *java virtuální stroj*. Ve windows se aplikace otevře pouhým dvojklikem na spustitelný soubor. V linuxu se aplikace spouští z příkazové řádky zadáním příkazu:

```
java -jar Gomoku.jar
```

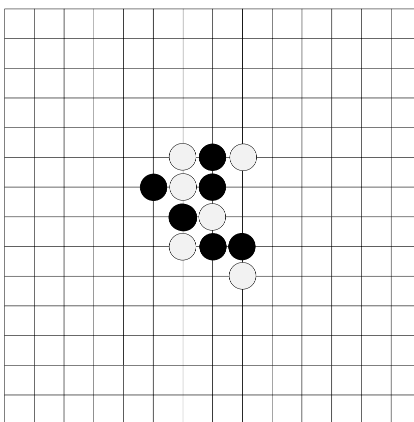
A.2 Zdrojové soubory

Ve složce Gomoku jsou uvedeny veškeré zdrojové soubory aplikace.

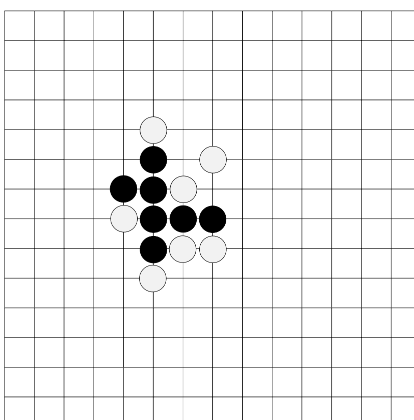
Dodatek B

Zkoumané herní situace

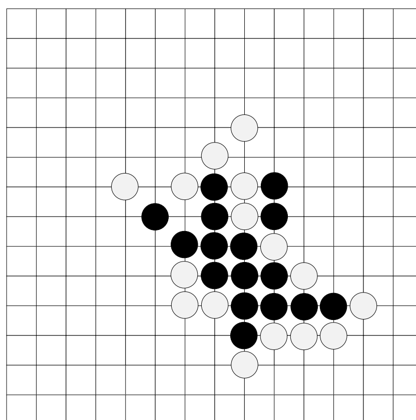
Na následujících herních situacích byly prováděny veškeré testy. Situace byly záměrně vybírány tak, aby se u nich měnily obranné a útočné strategie.



Obrázek B.1: **Situace č.1.** Na tahu je černý hráč a brání.



Obrázek B.2: **Situace č.2.** Na tahu je černý hráč a útočí.



Obrázek B.3: **Situace č.3.** Na tahu je černý hráč. Pro nižší hloubky prohledávání se ještě pokouší o další útok, pro vyšší hloubky prohledávání přechází na obranu.