PŘÍRODOVĚDECKÁ FAKULTA UNIVERZITY PALACKÉHO
KATEDRA INFORMATIKY

BAKALÁŘSKÁ PRÁCE

Monte Carlo vyhledávací techniky v deskových hrách

Radomír Škrabal

Anotace

Prohlašuji že jsem bakalářskou práci zpracoval samostatně pod vedením pana Mgr. Petra Osičky, Ph.D., s použitím uvedené literatury.

Děkuji svému vedoucímu bakalářské práce panu Mgr. Petru Osičkovi, Ph.D. za rady a pomoc při psaní této práce, dále děkuji panu Bc. Radku Janoštíkovi za poskytnutí přístupu k serveru tux pro mé experimenty a také všem kteří mě během psaní této práce podporovali.

Obsah

1. Úvod **6**
1.1. Shrnutí obsahu práce 6

2. Metody Monte Carlo a Monte Carlo Tree Search **7**
2.1. Metody Monte Carlo 7
2.2. Monte Carlo Tree Search 7

3. Hra THUD! **11**
3.1. Historie hry THUD! 11
3.2. Pravidla hry THUD! 11
3.3. Analýza složitosti hry THUD! 12

4. Implementace algoritmů **17**
4.1. Použité prostředky 17
4.2. Implementace pravidel 17
4.3. Algoritmus Monte Carlo 17
4.4. Pseudokód 18
4.5. Monte Carlo Tree Search 19
4.6. Pseudokódy 20
4.7. Návrhy vylepšení algoritmů 21
4.8. Pseudokódy Vylepšení 22

5. Experimenty **23**
5.1. Experiment - správnost algoritmu Monte Carlo 23
5.2. Experiment - Vylepšení algoritmu Monte Carlo zkrácením simulací . 24
5.3. Experiment - Vylepšení změnou simulační techniky 25
5.4. Experiment - síla algoritmu Monte Carlo 26
5.5. Experiment - správnost algoritmu Monte Carlo Tree Search 27
5.6. MCTS - Ladění konstanty 28
5.7. Experiment - Vylepšení MCTS zkrácením simulací 29
5.8. Experiment - Vylepšení MCTS změnou simulační techniky 29
5.9. Experiment - Síla algoritmu MCTS 30

6. Závěr **32**

Reference **32**

Seznam obrázků

1. Ilustrační obrázek k Integraci pomocí metody Monte Carlo 8
2. Jednotlivé fáze algoritmu Monte Carlo Tree Search 9
3. Hrací deska THUDu v počátečním stavu 12
4. Příklady tahů figurky trpaslíka 13
5. Příklady tahů figurky trolla 13
6. Histogram množství možných tahů v 5000 utkáních náhodného hráče 15

Seznam tabulek

1. Průměrná délka hry THUD! 14
2. Porovnání složitosti různých her s hrou THUD! 15
3. Tabulka výkonu algoritmu Monte Carlo pro různé zástupné hodnoty simulací 18
4. Výsledky experimentu funkčnosti algoritmu Monte Carlo. 23
5. Výsledky experimentu funkčnosti algoritmu Monte Carlo. 23
6. Výsledky experimentu funkčnosti vylepšení zkrácením simulací algoritmu Monte Carlo. 24
7. Výsledky experimentu délky zkrácení simulací algoritmu Monte Carlo. 24
8. Výsledky experimentu délky zkrácení simulací algoritmu Monte Carlo. 24
9. Výsledky experimentu délky zkrácení simulací algoritmu Monte Carlo. 25
10. Statistické hodnoty počtů simulací u jednotlivých hráčů během experimentu 25
11. Výsledky experimentu nahrazení náhodných simulačních hráčů Minimaxem hloubky 1 25
12. Výsledky experimentu nahrazení náhodných simulačních hráčů Minimaxem hloubky 2 26
13. Statistiky počtu simulací během výpočtu jednoho tahu u jednotlivých simulačních metod 26
14. Maximálně vylepšené Monte Carlo vs. Minimax hloubky 2 27
15. Výsledky experimentu funkčnosti algoritmu MCTS. 27
16. První experiment ladění konstanty MCTS 28
17. Druhý experiment ladění konstanty MCTS 28
18. Třetí experiment ladění konstanty MCTS 29
19. Experiment zkrácení simulací algoritmu MCTS 29
20. Výsledky experimentu nahrazení náhodných simulačních hráčů Minimaxem hloubky 1 30
21. Výsledky experimentu nahrazení náhodných simulačních hráčů Minimaxem hloubky 2 30
22. Výsledky experimentu síly algoritmu MCTS, proti algoritmu Monte Carlo. 30
23. Výsledky experimentu síly algoritmu MCTS, proti algoritmu Minimax. 31

Abstrakt

Tato práce představuje deskovou hru THUD! a analyzuje její složitost. Dále pro tuto deskovou hru navrhuje počítačové hráče založené na metodě Monte Carlo a algoritmu Monte Carlo Tree Search. Rovněž se zabývá implementací navržených metod, experimenty s nimi a porovnání s klasickými přístupy k vytváření počítačových hráčů.

1. Úvod

Výzkum rozhodovacích algoritmů pro hraní deskových her je v centru zájmu výzkumu umělé inteligence prakticky od jejího počátku. Přístup založený na teorii her a budování herního stromu a heuristického ohodnocení stavu pro výpočet nejvýhodnějšího tahu reprezentovaný zejména algoritmem Minimax a jeho vylepšením v podobě Alfa-Beta prořezávání se ukázal jako dostačující pro velkou část běžných deskových her jako například 3x3 piškvorky (Tic Tac Toe), dámu nebo šachy. Záhy však bylo zjištěny, že existují hry - byť stejné kategorie jako předchozí - pro které je klasický přístup obtížně, pokud vůbec použitelný. A to z důvodu buďto příliš velkého faktoru větvení herního stromu nebo neexistence přirozené ohodnocovací funkce pro stav hry. Nejznámější a zároveň zdaleka nejzkoumanější takovouto deskovou hrou je japonská desková hra Go, v jejímž případě bylo v současnosti dosaženo pouze částečných úspěchů například právě za pomoci metod Monte Carlo.

Výzkum metod Monte Carlo a Monte Carlo Tree Search se v současné době zaměřuje nejvíce právě na hru Go v její nejsložitější verzi (hrací deska 19x19 polí), méně pozornosti je potom věnováno dalším hrám jako například Shogi nebo Renju (viz. [12] [13]). Nicméně tyto metody byly s úspěchem, ať už úplným nebo částečným použity i v neklasických deskových hrách, ať už se jedná o hry kde nejsou hráčům známy všechna důležitá data o hře jako například poker, moderní deskové hry jako například Thurms and Taxis [1] či dokonce videohry v reálném čase jako například StarCraft [3]. Já jsem se rozhodl aplikovat tyto metody na klasickou deskovou hru, která je jak z pohledu vývoje umělé inteligence tak složiostní analýza prakticky neprozkoumaná a to na hru THUD!

V této práci se pokouším vytvořit počítačového hráče pro deskovou hru THUD! a to jak na základě klasického přístupu tak pomocí metod Monte Carlo a algoritmu Monte Carlo Tree Search. Výstupem této práce by měla být analýza složitosti hry THUD!, knihovna jež tuto hru reprezentuje a obsahuje několik různých typů počítačových hráčů a nakonec experimenty provedené na této knihovně pro porovnání síly jednotlivých přístupů k vytváření počítačového hráče.

1.1. Shrnutí obsahu práce

V kapitole 2. popíšu základní principy algoritmů Monte Carlo a Monte Carlo Tree Search a navrhnu některá vylepšení algoritmů.

V kapitole 3. představím deskovou hru THUD! Stručně zmíním její původ a historii, popíšu pravidla a nakonec provedu analýzu této hry z pohledu složitosti rozhodování a porovnám ji s dalšími podobnými deskovými hrami.

V kapitole 4. podrobněji popíšu způsob jakým byly algoritmy a jejich vylepšené verze implementovány, zmíním některé problémy, jež při implementaci nastaly a uvedu použité pseudokódy všech užitých a testovaných algoritmů.

V kapitole 5. budu postupně popisovat provedené experimenty a měřit výkon jednotlivých algoritmů, nejdříve algoritmů založených na metodě Monte Carlo, následně algoritmů Monte Carlo Tree Search a nakonec jejich Sílu porovnám s klasickým přístupem teorie her v podobě algoritmu Minimax.

2. Metody Monte Carlo a Monte Carlo Tree Search

V této se budu zaobírat metodou Monte Carlo, zmíním její historii, uvedu příklad jejího využití v matematické analýze a podrobněji popíšu fungování této metody a algoritmu na ní založeném pro deskové hry. Dále rozeberu a popíšu fungování algoritmu Monte Carlo Tree Search, jakožto vylepšení původní metody a kombinaci s přístupem budování herního stromu známého z teorie her.

2.1. Metody Monte Carlo

První předchůdci metod Monte Carlo se objevili již v 18. st. Konkrétně jde o úlohu nazvanou Buffonova jehla [5]. Moderní metody Monte Carlo poprvé představili Stanislaw Marcin Ulam a John von Neuman ve 40. letech 20. stol. [6] pro zkoumání průchodu neutronů různými materiály. Dnes mají tyto metody široké využití včetně výpočtu integrálů[4], řešení diferenciálních rovnic nebo v umělé inteligenci.

Všechny algoritmy používající metodu Monte Carlo jsou založeny na náhodném vzorkování nějakého prostoru (obvykle příliš velkého pro prohledání hrubou silou). Následně je nad takto odebranými vzorky proveden nějaký deterministický výpočet jež se snaží aproximovat výsledek který by byl získán hrubou silou. Je zřejmé, že větší počet vzorků znamená přesnější výsledek.

Metodu Monte Carlo lze použít například k výpočtu Rizmanova integrálu. [4] Mějme kladnou spojitou funkci jednoho argumentu na intervalu $[a, b]$. Rizmanův integrál, tedy obsah plochy pod křivkou funkce můžeme spočítat pomocí metody Monte Carlo tak, že křivku uzavřeme do obdélníku o stranách délky $x = b - a$ a y kde $y >= max(f)$ jehož obsah snadno spočítáme. Následně do tohoto obdélníků náhodně umísťujeme body a zaznamenáváme zda byl bod umístěn nad nebo pod křivku. Obsah plochy pod křivkou následně vypočteme jako:

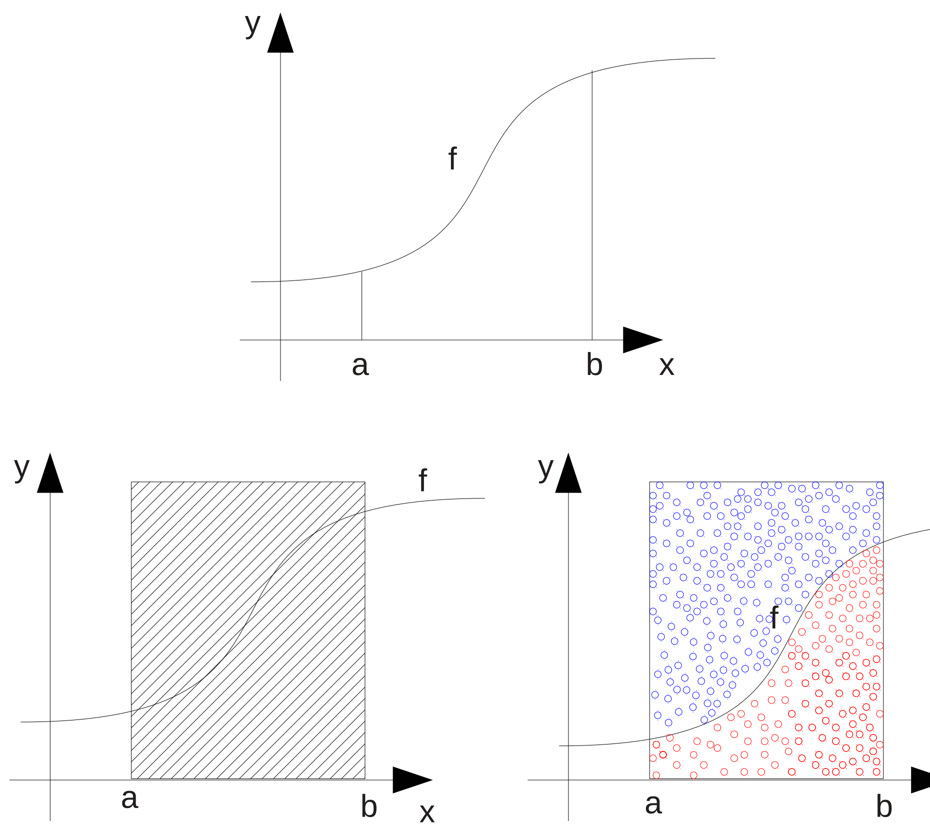
$$x * y * p$$

kde p je procentuální vyjádření počtu bodů, jež dopadly pod křivku. Ilustraci postupu můžete vidět na obrázku 1.

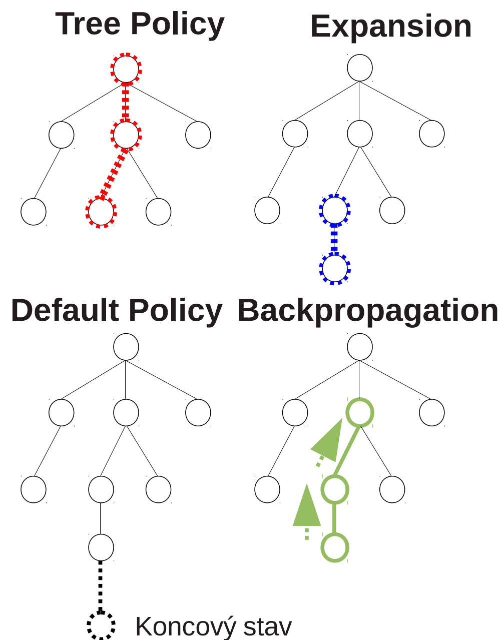
Pro výpočet nejlepšího tahu v deskových hrách se metoda chová následovně. Za prohledávaný prostor považujeme herní strom dané hry a ten prohledáváme z uzlu kde se nacházíme dále do hloubky. Algoritmus v každé iteraci svého hlavního cyklu vybere náhodný tah z množiny možných tahů a provede nad ním náhodnou simulaci. Toto vybrání náhodného tahu a následná simulace se považují za odebrání vzorku a dá se říci že se jedná o náhodný průchod herním stromem do hloubky. Toto se provádí dokud nedosáhneme nějakého (obvykle časového) limitu. Jakmile skončí fáze vzorkování proběhne deterministický výpočet, což znamená, že z tahů se podle k nim příslušných ohodnocení vzorků vybere ten nejslibnější tah, tedy tah pro nějž vycházely simulace nejlépe. Rozbor konkrétních metod výběru tahu na základě vzorků můžete najít v kapitole 4.

2.2. Monte Carlo Tree Search

Metoda Monte Carlo Tree Search v sobě kombinuje přístupy budování herního stromu z algoritmů typu Minimax a náhodné vzorkování algoritmů založených na metodě Monte Carlo. Algoritmus při svém běhu postupně pomocí náhodných simulací dalšího průběhu hry buduje co nejslibnější podmnožinu herního stromu. V každé iteraci hlavního cyklu algoritmu proběhnou čtyři hlavní fáze (viz obrázek 2.) fáze Tree Policy – fáze průchodu stromem do hloubky, fáze Expansion – fáze rozšíření stromu, fáze Default policy – ohodnocení uzlu pomocí simulace a Backpropagation



Obrázek 1. Ilustrační obrázek k Integraci pomocí metody Monte Carlo



Obrázek 2. Jednotlivé fáze algoritmu Monte Carlo Tree Search

– aktualizace ohodnocení dosud vybudované části stromu. Dále tyto fáze rozebírám podrobněji.

Tree Policy Fáze tree policy (někdy rovněž nazývaní Selection [1]) se v algoritmu Monte Carlo Tree Search stará o částečné budování herního stromu. Dosud vybudovaná část herního stromu je procházena do hloubky dokud není nalezen uzel vhodný k rozšíření (další fáze). Tato fáze rovněž řeší takzvané Exploration Explotation dilema, kde se při průchodu do hloubky rozhoduje zda zvolit doposud nejslibnější podstrom (exploitation), nebo prozkoumávat méně vyčerpané a potenciálně slibné podstromy (exploration). Jednoduše řečeno chování typu exploitation nám upřesňuje hodnotu dosud nejlepšího tahu, kdežto exploration se snaží najít tah eventuálně optimálnější. Důkladněji tento problém rozebírám dále.

Expansion Fáze expansion dále rozšiřuje částečně vybudovaný herní strom o nové listy. Takto přidaných listů může být libovolné množství, často se ale přidává pouze jeden náhodně vybraný.

Default Policy Default policy (někdy rovněž Simulation) označuje fázi náhodného průchodu herním stromem, obvykle do koncového stavu hry, kterým algoritmus získá nějaké číselné ohodnocení stavu ze kterého simulace vycházela. Takovéto simulace probíhají v podstatě stejně jako u základní verze algoritmu Monte Carlo.

Backpropagation V této fázi se využijí data (tedy číselné ohodnocení) získaná v předchozím kroku pro upřesnění hodnot v částečně vybudovaném herním stromu. Z naposledy přidaného (a tedy v daném kroku simulovaného) listu se prochází strom

směrem nahoru, přes rodiče uzlů a na základě hodnoty ze simulace se aktualizují jejich hodnoty.

Koncové vybrání nejlepšího tahu Fáze Tree Policy, Expansion, Default Policy a Backpropagation probíhají po určitou dobu (algoritmus je stejně jako Monte Carlo omezen nějakým zdrojem, obvykle časem) v hlavním cyklu algoritmu. Jejich mnohonásobnou aplikací, dostaneme částečně vybudovaný herní strom ve kterém by měly být nejvíce rozvinuty a prozkoumány ty nejoptimálnější podstromy. V tuto chvíli je tedy nutné vybrat co nejlepšího potomka kořenu tohoto stromu. To lze obecně udělat s ohledem na několik různých kritérií [7]. Můžeme vybrat nejlépe ohodnoceného potomka (Max Child), nejrobustnějšího potomka (Robust Child), nejlépe ohodnoceného a nejrobustnějšího potomka (Max-Robust Child), nebo tzv. Bezpečného potomka (Secure Child), tedy potomka který maximalizuje tzv. Lower Confidence Bound, hodnotu zavedenou pro uzel n jako $value(n) + \frac{P}{\sqrt{visitCount(n)}}$ [7], kde $value(n)$ je hodnota uzlu, $visitCount(n)$ je počet návštěv uzlu n během výpočtu a P je parametr.

UCT strategie V algoritmu, konkrétně v jeho fázi Tree Policy je nutné řešit exploration exploitation dilema, tedy rozhodovat se jestli při průchodu stromem do hloubky budeme volit nejlépe ohodnocené podstromy nebo budeme hledat lepší řešení v méně prozkoumaných podstromech. Tento problém je velmi podobný tzv. Multi-armed bandit problému. V něm, jde o to získat co největší množství peněz ze skupiny výherních automatů s náhodným rozdělením odměn se stanoveným základním obnosem peněz. Na základě tohoto problému navrhli Kocsis a Szepesvári [2] takzvanou výběrovou strategii UCT pro stromy. Tento přístup hledá takový uzel n aby maximalizoval následující výraz:

$$\frac{value(n)}{visitCount(n)} + c \cdot \sqrt{\frac{2ln(visitCount(p)}{visitCount(n)}}$$

Kde n je uzel pro něhož hledáme hodnotu, p je předek tohoto uzlu, $value$ je hodnota přiřazená uzlu a $visitCount$ je počet prohledávání jež už byl v daném uzlu proveden a c je konstanta. Povšimněme si, že výraz se skládá ze dvou fundamentálních částí. První zlomek ve výrazu je přímo úměrný hodnotě přiřazené uzlu a naopak nepřímo úměrný tomu kolikrát jsem uzlem při prohledávání již prošli, tato část zlomku tedy reprezentuje chování typu exploitation, tedy snahu co nejvíce vytěžit v danou chvíli nejlepší řešení. Naopak druhý sčítanec nabývá vyšších hodnot zejména pro uzly jež samy o sobě nebyly moc prozkoumány, ale jejich předek byl naopak prozkoumán hodně. Toto reprezentuje chování typu exploration, snahu prohledávat sub-optimální řešení. U této části výrazu navíc figuruje konstanta c, ta umožňuje v algoritmu řídit množství průzkumu na úkor vytěžovaní a naopak. Tuto konstantu je pro danou hru nutné odladit a z toho se i sestává důležitá část vývoje algoritmu, viz experimenty v kapitole 5.

3. Hra THUD!

V této části podrobněji rozebereme deskovou hru THUD! Krátce shrneme jení původ a historii, podrobně popíšeme standardizovaná pravidla a provedeme analýzu hry z hlediska její stavové složitosti a velikosti herního stromu.

3.1. Historie hry THUD!

Původ hry THUD! tkví v literárním světě Zeměplochy (Discworld) vymyšleného sirem Terry Pratchetem. Hra vznikla v roce 2002 jako součást rozšiřující se značky Discworld a byla inspirována reáliemi knih a světa Zeměplochy. Hru THUD! Vymyslel Trevor Truran [8] který založil její pravidla na starých severských deskových hrách jako Hnefatafl a Tablut [8], upravil je však tak aby pravidla byla méně jednostranná. V románech sira Terryho Pratcheta se hra nejvíce zviditelnila ve stejnojmenném románu THUD! (2005 Doubleday, česky BUCH! 2005 Talpress). Dnes okolo této deskové hry existuje široká komunita, pořádají se turnaje a vznikají další varianty hry (Koom valley THUD!)

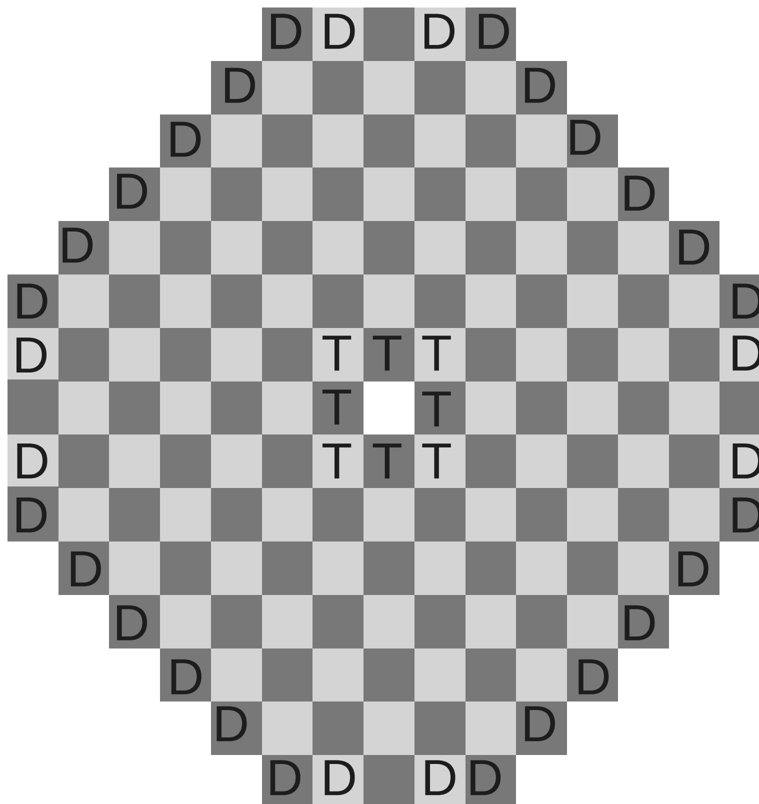
3.2. Pravidla hry THUD!

Hra THUD! Má následující pravidla:

- Hru THUD! hrají dva hráči.
- Utkání hry THUD! Se skládá ze dvou samostatných jednoduchých her, kde si ve druhé hře hráči vymění strany.
- V každé jednoduché hře hraje jeden hráč za stranu trpaslíků a druhý za stranu trollů.
- První tah má vždy strana trpaslíků, hráči se na tahu střídají.
- Utkání vyhrává hráč jež má vyšší celkové skóre.
- Skóre se počítá dle počtu zbývajících figurek hráče a to 1 bod za každého zbylého trpaslíka a 4 body za každého zbylého trolla.
- Jedna z her utkání je ukončena když se oba hráči dohodnou, že již není možné provést žádná zajmutí nepřátelských figurek.
- Hrací deska je osmiúhelník o straně 5 polí s nepřístupným, nepohyblivým ústředním kamenem, (viz obrázek 3.) má tedy celkem 164 polí. Na začátku jednoduché hry je na okrajích desky rozestavěno 32 trpaslíků a okolo ústředního kamene 8 trollů.

Ve hře THUD! se vyskytují následující figurky a táhnou dále popsaným způsobem:

- Trpaslíci: Trpaslíci se běžným způsobem pohybují jako dáma v šachu. Pro zajímání nepřátelských figurek (trollů) musí provést speciální pohyb tzv. "Hurling“. Hurling probíhá následovně: Pokud trpaslíci vytvoří na hrací desce nepřerušenou řadu (horizontální, vertikální, popř. libovolně diagonální) mohou trpaslíka na okraji řady vrhnout až o tolik polí vpřed jaká je délka řady, ale to pouze za předpokladu, že se na cílovém poli vrhu nachází troll a mezi cílovým a startovním polem není na přímce žádná figurka ani ústřední kámen. Tento troll je takto zajat a odebrán z hrací desky. Vržený trpaslík poté obsadí jeho místo. Každý figurka trpaslíka sama o sobě tvoří řadu délky 1. (příklad můžete vidět na obrázku 4.)



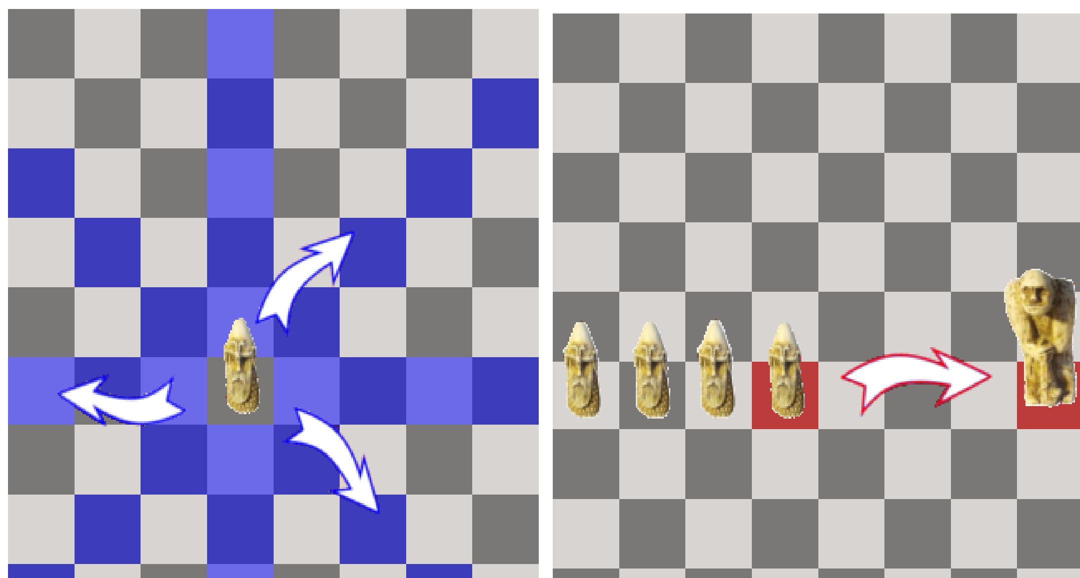
Obrázek 3. Hrací deska THUDu v počátečním stavu

- Trollové: Trollové se běžným způsobem pohybují jako král v šachu. Vždycky když je troll přesunut na nějaké pole (i během Shovingu, viz dále) automaticky zajme všechny trpaslíky na všech osmi polích okolo pole kam troll dopadl. Dále mají trollové speciální pohyb nazvaný "Shoving". Pokud trollové vytvoří nepřerušenou řadu (horizontální, vertikální, popř. libovolně diagonální) mohou trolla na okraji řady strčit až o tolik polí vpřed jaká je délka nepřerušené řady trollů, ale pouze za předpokladu, že pole na které troll dopadne je prázdné a mezi cílovým a startovním polem není na přímce žádná figurka ani ústřední kámen. (příklad můžete vidět na obrázku 5.)

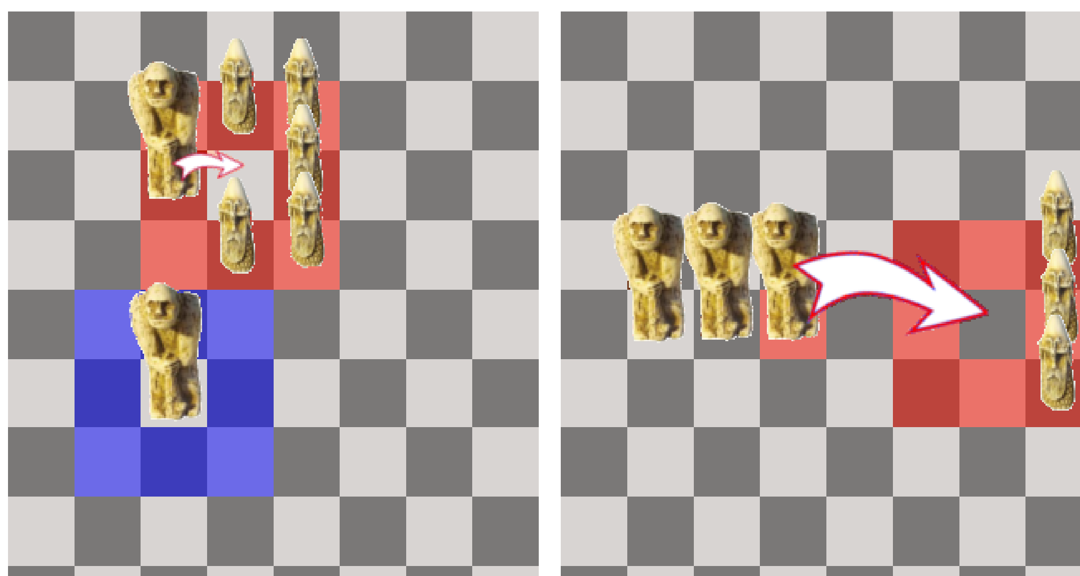
3.3. Analýza složitosti hry THUD!

Vzhledem k počtu figurek na hrací ploše a jejich možnostem tahů lze odhadovat, že hra THUD! bude mít jak vysokou stavovou složitost (game state complexity) tak velký herní strom (game-tree complexity).

Hrací deska má tvar osmiúhelníku vepsaného čtverci o straně patnáct polí. Počet platných polí na kterých se mohou figurky během hry dostat dostaneme odečtením čtyř neplatných rohů a ústředního kamene od obsahu čtverce 15 x 15, tedy:



Obrázek 4. Příklady tahů figurky trpaslíka



Obrázek 5. Příklady tahů figurky trolla

Hráč	Průměrná délka hry
RandomAI	277
MinimaxAI 1	175
MinimaxAI 2	291

Tabulka 1. Průměrná délka hry THUD!

$$(15 \cdot 15) - 4 \cdot 15 - 1 = 225 - 61 = 164$$

Vzhledem ke způsobu jakým mohou figurky na hrací ploše táhnout, lze s jistotou říci, že každá figurka se může během hry dostat na kterýkoli z platných polí hrací plochy. Díky tomu můžeme pro výpočet stavové složitosti hry THUD! Požít jednoduchou kombinatoriku, kde nejdříve umístíme na hrací plochu všechny figurky trpaslíků a následně všechny figurky trollů a to opakujeme pro postupně se snižující počty figurek na obou stranách. Tím tedy dostáváme vztah:

$$\sum_{x=31}^{0}\sum_{y=7}^{0}(\prod_{i=0}^{x}(164-i)\cdot(\prod_{j=0}^{y}(164-i-j))) = 4,08217 \cdot 10^{90}$$

Stavová složitost hry THUD! Je tedy $4,08 \cdot 10^{90}$.

Dalším krokem při určování složitosti hry THUD! je výpočet složitosti herního stromu. Složitost herního stromu je určena počtem listů v plně rozvinutém rozhodovacím stromu s kořenem v počátečním stavu hry. Výpočet takovéhoto stromu je však příliš náročný, spokojím se proto - jak je běžná praxe - s hrubým horním odhadem složitosti herního stromu za pomoci průměrného faktoru větvení a průměrné délky hry.

Za tímto účelem byl proveden experiment pro určení těchto údajů. Nejdříve byla sledována délka hry THUD! pro 3 různé počítačové hráče v zápase sám se sebou, bylo odehráno 5000 utkání pro náhodného hráče, 5000 utkání pro algoritmus Minimax hloubky 1 a 1000 utkání pro algoritmus minimax hloubky 2. Průměrné délky hry můžete vidět v tabulce 1.

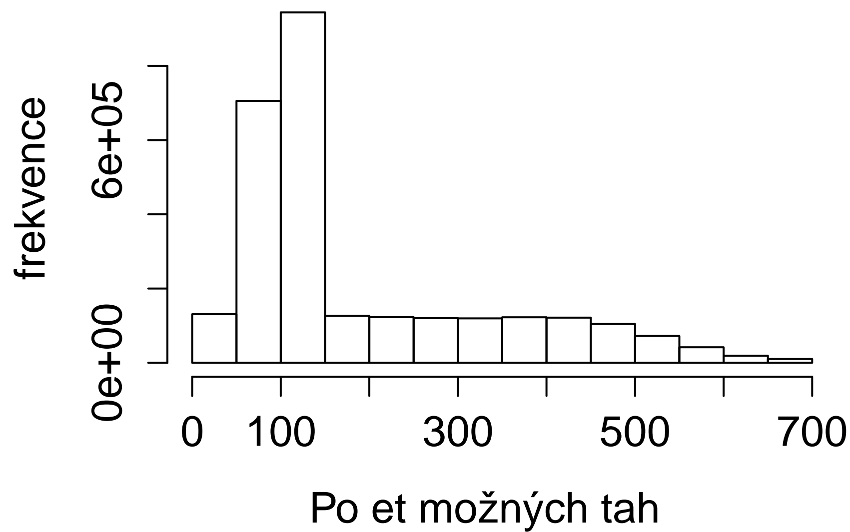
Vzhledem k tomu, že Minimax hloubky 2 je zřejmě nejsilnější z testovaných hráčů bude pravděpodobně i nejlépe aproximovat skutečné hodnoty délek jednoduchých her THUDu! Zde je nutné upozornit na neobvyklý způsob ukončovacího pravidla jednoduché hry (viz. 3.2.) kde je nutná dohoda obou hráčů nebo anihilace jednoho z nich a to vytváří značně dlouhé hry, zvláště pro počítačové hráče (viz. 4.2.), proto je nutné naměřené délky her brát s rezervou.

Dále byl proveden experiment za účelem zjištění průměrného faktoru větvení ve hře THUD! V experimentu hrál náhodný hráč sám proti sobě 5000 utkání v jejichž celém průběhu byla sledována mohutnost množiny možných tahů. Zde je nutné zmínit, že vzhledem k asymetrii herních stran má tato veličina tendenci nabývat jak relativně nízkých (v maximu okolo 64 pro trolly), tak i značně vysokých (přes 650 v maximu pro trpaslíky) hodnot. Průměr naměřených hodnot byl 188.59. Na histogramu 6. si můžete prohlédnout rozložení naměřených hodnot.

Ze získaných dat můžeme tedy získat hrubý horní odhad složitosti herního stromu hry THUD! tak, že umocníme naměřenou průměrnou délku hry průměrným počtem možných z čehož dostáváme:

$$291^{189} = 8.59 \cdot 10^{423}$$

Zjistili jsme tedy stavovou složitost a průměrný faktor větvení hry THUD! Pokud tato čísla porovnáme s jinými deskovými hrami ([9] [10] [11]) můžeme vidět, že



Obrázek 6. Histogram množství možných tahů v 5000 utkáních náhodného hráče

Hra	Stavová složitost	Horní odhad složitosti herního stromu
Awari	10^{12}	10^{32}
Spoj čtyři	10^{14}	10^{21}
Dáma	10^{21}	10^{31}
Šachy	10^{46}	10^{123}
Čínské šachy	10^{48}	10^{150}
Shogi	10^{71}	10^{226}
Go-Moku(15x15)	10^{105}	10^{70}
Renju(15x15)	10^{105}	10^{70}
Go(19x19)	10^{172}	10^{360}
THUD!	10^{90}	10^{423}

Tabulka 2. Porovnání složitosti různých her s hrou THUD!

složitost hry THUD! je značně vysoká, z tabulky 2. můžeme vyčíst, že její stavová složitost překonává Japonské šachy Shogi a horní odhad složitosti herního stromu hry THUD! dokonce značně překonává i Go na největší desce (19x19). Nicméně je nutné brát v potaz to, že horní odhad složitosti herního stromu může být kvůli statistické chybě značně nepřesný a tento závěr je tedy nutné brát s rezervou.

4. Implementace algoritmů

Tato kapitola rozebírá implementaci a prostředky implementace pravidel hry THUD! a algoritmů Monte Carlo a MCTS jež byly použity v experimentech popsaných v kapitole 5.

4.1. Použité prostředky

Implementace veškerých dále popsaných algoritmů, pravidel a funkcí byla provedena v programovacím jazyku Java verze 1.6. Byly použity pouze standardní knihovny jazyka. Pro dále popsané omezení výpočtu algoritmů Monte Carlo a MCTS byly použity metody tříd java.lang.management.ManagementFactory a java.lang.management.ThreadMXBean které dokážou měřit čas kódu strávený na procesoru.

4.2. Implementace pravidel

Implementace samotných pravidel hry THUD! Byla až na části ukončení jednoduché hry a rozlišování jednoduché hry a utkání zcela přímočará, budu zde tedy podrobněji popisovat pouze tyto dvě části implementace.

Ukončení Důležitým bodem je implementace ukončení jednoduché hry. To je v pravidlech hry THUD! Specifikováno jako dohoda dvou hráčů o tom, že již nemůže být zajata žádná z figurek na hrací ploše. Toto pravidlo bylo implementováno všem algoritmům stejně a to následovně:

Hráč je ochoten ukončit hru právě když je splněna některá z následujících podmínek:

- Hráč v současném stavu vyhrává (má kladné skóre)
- Hráč prohrál (byl anihilován, nezbyla mu žádná herní figurka)
- Hráč po určitý stanovený počet tahů n nebyl schopen provést zajmutí nepřátelské figurky

Jednoduchá hra je tedy ukončena, když jsou oba hráči ochotni se vzdát. Parametr n byl pro všechny počítačové hráče nastaven jako 120 tahů.

Utkání vs. Jednoduchá hra Každé utkání deskové hry THUD! Se ve skutečnosti skládá ze dvou "jednoduchých her", kde první jednoduchou hru hrají hráči klasicky a ve druhé hře si vymění strany. Toto bylo implementováno podle pravidel, každá z her v mé implementaci se tedy skládá ze dvou jednoduchých her. Samotná implementace sice umožňuje toto pravidlo ignorovat, tato vlastnost je zde ale pouze pro ladící účely a při ostrém běhu se nepoužívat. Všechny experimenty v kapitole V. Jsou prováděny nad hrami podle pravidel, tedy včetně výměny stran a druhé hry.

4.3. Algoritmus Monte Carlo

Všechny algoritmy využívající metodu Monte Carlo jsou založeny na náhodném vzorkování prohledávaného prostoru. Prostor je obvykle příliš velký na to aby mohl být prohledán hrubou silou celý, ale není to podmínka. Konkrétně u deskových her "ploché Monte Carlo" (flat Monte Carlo viz [3]) funguje tak, že z daného stavu je vygenerována množina možných akcí (za prohledávaný prostor v tomto případě považujeme herní strom) a z té je poté odebráno určité množství vzorků (to jaké množství vzorků je záleží na omezení výpočtu algoritmu. Obecně je možné několik

Utkání	výhry 1. hráče	remízy	výhry 2. hráče
Medián vs. Mean	91.6%	4.4%	4.0%
Modus vs. Medián	6.8%	6.4%	86.8%
Modus vs. Mean	41.6%	21.6%	36.8%

Tabulka 3. Tabulka výkonu algoritmu Monte Carlo pro různé zástupné hodnoty simulací

řešení). Vzorek je odebrán tak, že je vybrán náhodný tah z množiny možných tahů a je s ním provedena simulace. Nad takto odebranými vzorky je proveden deterministický výpočet, kdy je ze vzorků vybrán ten který se nějakým způsobem ukázal jako nejlepší.

Je nutné podotknout, že algoritmu nic nebrání nějaký vzorek odebrat dvakrát nebo vícekrát. Takto získáme "tabulku" odebraných vzorků s přiřazenými výsledky jejich simulací. Deterministický výpočet tedy musí nějakou statistickou metodou vybrat vzorek s největším potenciálem. Protože jeden tah může mít obecně více simulací a tedy hodnot, je nutné mu nějakým způsobem přiřadit jednu hodnotu. Nabízí se metoda aritmetického průměru, mediánu nebo modu. Pro určení nejvhodnější metody jsem algoritmus s různými způsoby výběru nejlepšího vzorku nechal hrát proti sobě. Jak je vidět z tabulky 3. nejsilnějším způsobem výběru se ukázal být medián.

Důvod je ten, že simulace jsou zcela náhodné a proto v nich často vzniknou odlehlé hodnoty, které pravděpodobně v dalším průběhu hry ve skutečnosti nenastanou a proto je aritmetický průměr, jenž je značně citlivý na takovéto odlehlé hodnoty jako zástupná hodnota nevhodný. Modus je jak se zdá výhodnější, ale vzhledem k vysokému faktoru větvení hry THUD! a ohodnocení nacházejícímu se v poměrně širokém rozmezí -32 až 32 je poměrně malá pravděpodobnost, že se nějaká hodnota bude v odebraných vzorcích vyskytovat vícekrát a pokud ano tak, že to bude hodnota jež bude vhodná jako zástupná hodnota všech vzorků. Naopak medián není ani citlivý na odlehlé hodnoty ani nevyžaduje aby byla jeho hodnota zastoupena v simulacích mnohokrát, je proto vhodný jako zástupná hodnota pro množinu vzorků.

Jak jsem zmínil výše počet vzorků lze omezit několika způsoby, nabízí se možnost nastavit fixní počet vzorků, uzpůsobit počet vzorků velikosti množiny (tedy uzpůsobit počet vzorků jako poměr vzhledem k možným akcím z daného stavu) nebo omezit algoritmus časem. Poslední možnost je nejpoužívanější a zřejmě nejlepší. Síla algoritmu Monte Carlo je zřejmě přímo závislá na tom kolik vzorků odebere a kolik simulací provede a omezení časem dovoluje snadno sílu algoritmu škálovat. Konkrétně je v implementaci omezen čas jež stráví hlavní cyklus algoritmu na procesoru pomocí prostředků popsaných v 4.1.

Simulace v rámci algoritmu byly prováděny počítačovými hráči stejnými jako ti jež byly použiti v experimentech v V. Pokud šlo o nevylepšený algoritmus Monte Carlo jednalo se o dva náhodné hráče, kde byla simulovaná hra ukončena za stejných podmínek jako by to byla běžná jednoduchá hra.

4.4. Pseudokód

Následují komentované pseudokódy funkcí algoritmu Monte Carlo.

Funkce MonteCarlo Funkce MonteCarlo pomocí metody Monte Carlo vypočte a navrátí tah určený k zahrání.

V parametrech dostane funkce stav hry (parametr state) a hráče pro kterého

má tah vybírat. V proměnné moves se nachází množina všech možných tahů z daného herního stavu a proměnná resultTable reprezentuje strukturu pro uložení provedených simulací. V hlavním cyklu algoritmu potom zvolíme náhodný tah z množiny a provedeme nad ním simulaci (viz funkce simulate). Výsledek simulace uložíme do tabulky.

Jakmile spotřebujeme přidělený čas pro výpočet projdeme tabulku simulací. Pro každý ze simulovaných tahů vypočteme medián výsledků jeho simulací. Z takto získaných ohodnocení vybereme to nejvyšší a asociovaný tah použijeme jako návratovou hodnotu funkce.

```
function monteCarlo(state, player)
  moves ← getMoves(state, player)
  resultTable ← initResultTable(Move, List)
  while CPUtimeNotDepleted do
    move ← chooseUniformlyAtRandom(moves)
    score ← simulate(cpyState(state), player, move)
    ResultTable.add(move, score)
  end
  return getMoveWithBestModus(resultTable)
endfunction
```

Funkce Simulate – základní verze Funkce Simulate provede nad daným herním stavem a tahem náhodnou simulaci zahrání tohoto tahu a vrátí její výsledek. (bodové ohodnocení desky po ukončení simulace podle pravidel hry viz 3.2.)

V parametrech dostane funkce herní stav, ze kterého se bude simulace provádět, tah jehož vliv se má simulovat a hráče příslušného tomuto tahu.

Nejdříve je zahrán tah předaný v parametru. V cyklu while je náhodně vybrán tah příslušný danému hráči a ten je zahrán na desku. To se děje tak dlouho doku podle pravidel není ukončena jednoduchá hra (viz III. b.). Funkce vrací ohodnocení stavu po ukončení hry podle pravidel (3.2.).

```
function simulate(state, player, move)
  state.applyMove(move)
  switchPlayer(player)
  while state is nonterminal do
    moves ← getMoves(state, player)
    move ← chooseUniformlyAtRandom(moves)
    state.applyMove(move)
    switchPlayer(player)
  end
  return getStateScore(state)
endfunction
```

4.5. Monte Carlo Tree Search

Algoritmus Monte Carlo Tree Search v sobě kombinuje vlastnosti náhodného vzorkování a budování částečného herního stromu. V každé iteraci hlavního cyklu algoritmu se provedou čtyři hlavní operace [3]:

- **Tree Policy** Tree Policy projde doposud vybudovanou část částečného herního stromu, do hloubky pomocí vztahu pro Exploration Explotation dilema (viz. [3]) a vybere uzel určený pro rozšíření.
- **Expansion** Rozšíří uzel vybraný pomocí Tree Policy o náhodně vybraného potomka.

- **Default Policy** Provede průchod herním stromem do hloubky až do koncového stavu.
- **Backpropagation** S pomocí ohodnocení koncového stavu získaného v Default Policy provede aktualizaci ohodnocení uzlů v částečně vybudovaném herním stromu.

Tedy v každé iteraci dojde k přidání jednoho uzlu a aktualizaci ohodnocení předků tohoto uzlu. Jako vybraný tah se poté použije nejlépe ohodnocený potomek kořene částečně vybudovaného herního stromu.

U fáze průchodu dosud vybudovanou část stromu (Tree Policy) se algoritmus rozhoduje kterou z větví bude prozkoumávat. Toto rozhodování zajišťuje funkce BestChild. Podrobněji o tomto problému jsem psal v kapitole 2.2..

Fáze Default Policy je obdobou procesu simulace z algoritmu Monte Carlo. Lze pro ni použít stejné prostředky i vylepšení.

Fáze Backpropagation aktualizuje herní strom, musí tedy brát v potaz různé hráče, jež byli v daném uzlu stromu na tahu a přidělovat na základě předchozí simulace adekvátní ohodnocení. Vzhledem k tomu, že hru THUD! hrají vždy pouze dva hráči je možné použít zjednodušenou verzi této aktualizace založenou na algoritmu negamax jak navrhuje [3].

4.6. Pseudokódy

Následují pseudokódy funkcí algoritmu MCTS.

Funkce MCTS vybere na základě metody Monte Carlo Tree Search tah. V parametrech dostane funkce stav hry a hráče na tahu. Proměnná tree označuje částečně vybudovaný herní strom, proměnná sample označuje nově přidaný list stromu určený k simulaci.

```
function MCTS(state, player)
  tree ← initializeMCTSTree(state, player)
  while CPUTimeNotDepleted do
    sample ← selectNode(tree) //Tree Policy
    score ← simulate(sample) //Default Policy
    backpropagate(sample, score) //Backpropagation
  end
  return bestChild(root(tree), 0)
endfunction
```

Funkce SelectNode vykonává krok popsaný v algoritmu jako Tree Policy.

```
function selectNode(tree)
  node ← root(tree)
  while node isNotTerminal do
    if not isFullyExpanded(node) then
      return expand(node)
    else
      node ← bestChild(node, Cp)
    endif
  end
endfunction
```

Funkce expand vykonává fázi expansion v algoritmu MCTS.

```
function expand(node)
  move ← selectFromUntriedMovesUniformlyAtRandom(node)
  child ← addChild(node, move)
  return child
endfunction
```

Funkce Backpropagate vykonává fázi Backpropagation v algoritmu MCTS.

```
function backpropagate(node, score)
  while node != null do
    visitCount(node)++
    setValue(node) = value(node) + score
    Score = −score
    Node = parent(node)
  end
endfunction
```

Funkce BestChild je funkce řešící Exploration Explotation dilema. V argumentech dostane funkce uzel herního stromu a konstantu c označující míru prozkoumávání. Funkce projde všechny potomky předaného uzlu a pro každý vypočte hodnotu výrazu pro exploration Explotation dilema. Nakonec vrátí takového potomka pro kterého měl vypočtený výraz nejvyšší hodnotu.

```
function bestChild(node, c)
  return argmax_{n∈potomci(node)}(value(n)/visitCount(n) + c · sqrt(2ln(visitCount(node))/visitCount(n)))
endfunction
```

$$argmax_{n \in potomci(node)}\left(\frac{value(n)}{visitCount(n)} + c \cdot \sqrt{\frac{2ln(visitCount(node))}{visitCount(n)}}\right)$$

4.7. Návrhy vylepšení algoritmů

Na oba algoritmy lze aplikovat různá vylepšení, jejich efektivita se ale často liší podle hry na kterou jsou algoritmy aplikovány. Experimenty a analýzu úspěchu/-neúspěchu zde navrhovaných vylepšení naleznete v Kapitole 5.

Zkrácení Simulací Efektivita jak algoritmu Monte Carlo tak Monte Carlo Tree Search přímo závisí na tom kolik simulací dokáží během své přidělené výpočetní doby (viz 2.1. a 2.2.) provést. Pokud tedy nechceme zvyšovat čas přidělený algoritmu na procesoru nabízí se možnost nějakým způsobem zrychlit, nebo zjednodušit samotnou simulaci. Zde navrhovaným vylepšením je zkrácení doby jedné simulace tím, že nebudeme provádět simulaci až do koncového stavu hry (tedy listu herního stromu), ale budeme simulovat hru jen po určitý, pevně daný počet tahů. Toto vylepšení by tedy mělo výrazně zvýšit počet simulací provedených algoritmem během jeho výpočetního času a tím i zvýšit jeho účinnost.

Pseudonáhodné Simulace Problémem náhodných simulací u obou algoritmů je jejich nepřesnost. Vzhledem k vysokému průměrnému faktoru větvení hry THUD! Mohou náhodné simulace často vracet hodnoty, jež jsou pro příslušný herní podstrom naprosto netypické. Tento problém částečně řeší velké množství simulací s pomocí mediánu jako zástupné hodnoty množiny jejich výsledků. Nicméně je i možné nesimulovat rychlý průchod herním stromem zcela náhodně, ale použít nějaký složitější algoritmus, který ale bude lépe aproximovat chování jednotlivých hráčů. To sice způsobí menší počet simulací, nicméně to zvýší jejich přesnost. Experimenty pro

použití jednotlivých simulačních postupů pro jednotlivé strany naleznete v kapitole 5.

4.8. Pseudokódy Vylepšení

Následují pseudokódy jednotlivých vylepšení. Vylepšení se pro oba algoritmy implementovaly stejně, konkrétně ve funkci simulate.

Funkce Simulate – Simulation Lenght Cut Tato verze funkce simulate se používá ve variantě algoritmu s vylepšením zkrácením simulací. Funkce pracuje stejně jako základní varianta, pouze hlavní cyklus se zastaví již po zadaném počtu simulací.

```
function simulateSimLenCut(state, player, move, simCut)
  state.applyMove(move)
  switchPlayer(player)
  for i from 0 to simCut do
    moves ← getMoves(state, player)
    move ← chooseUniformlyAtRandom(moves)
    state.applyMove(move)
    switchPlayer(player)
  end
  return getStateScore(state)
endfunction
```

Funkce Simulate – Pseudorandom simulation Tato verze funkce simulate využívá k simulaci hráčů nějakého jiného počítačového hráče. Takovýto počítačový hráč může být libovolný, tedy i náhodný. Pokud jsou oba počítačoví hráči náhodní přechází toto vylepšení ve standardní simulaci Monte Carlo.

```
function simulatePrandom(state, player, move, dwarfAI, trollAI)
  state.applyMove(move)
  switchPlayer(player)
  while state is nonterminal do
    moves ← getMoves(state, player)
    if player is DwarfPlayer then
      move ← dwarfAI.chooseMove(moves)
    endif
    if player is TrollPlayer then
      move ← trollAI.chooseMove(moves)
    endif
    state.applyMove(move)
    switchPlayer(player)
  end
  return getStateScore(state)
endfunction
```

Simple Monte Carlo (1s) vs. Random AI	
Výher	96.6%
Proher	1.4%
Remíz	2.0%

Tabulka 4. Výsledky experimentu funkčnosti algoritmu Monte Carlo.

Simple Monte Carlo (5s) vs. Random AI	
Výher	99.0%
Proher	0.0%
Remíz	1.0%

Tabulka 5. Výsledky experimentu funkčnosti algoritmu Monte Carlo.

5. Experimenty

V této kapitole budu prezentovat výsledky experimentů navržených pro všechny výše zmíněné algoritmy a jejich vylepšení. Nejdříve probereme výsledky algoritmů Monte Carlo a navrhovaných vylepšení. Budu sledovat sílu algoritmu při použití těchto vylepšení a snažit se získat co nejefektivnější algoritmus. Ten potom porovnám s klasickými postupy pro vytváření umělé inteligence, konkrétně s algoritmem minimax hloubky 2. Dále přijde na řadu algoritmus Monte Carlo Tree Search, odladění výkonu jeho základní verze a následných vylepšení. Nakonec otestuji jeho nejsilnější verzi v utkání s nejsilnějším získaným algoritmem Monte Carlo a rovněž s Minimaxem hloubky 2.

Ve všech experimentech jsou uvedena důležitá nastavení testovaných algoritmů. Pokud proti sobě byli v experimentu postaveni dva počítačový hráči typu Monte Carlo nebo Monte Carlo Tree Search, pak pokud není výslovně uvedeno jinak měly oba algoritmy na rozhodnutí stejně dlouhý čas a to 1 sekundu.

Všechny experimenty byly provedeny na serveru TUX, studenském serveru katedry informatiky Univerzity Palackého v Olomouci. V době provádění experimentů měl server následující konfiguraci 2 DualCore Xeons, 6GB RAM, 2TB RAID 5 a běžel pod operačním systémem Gentoo GNU/Linux. Bližší informace na http://tux.inf.upol.cz.

5.1. Experiment - správnost algoritmu Monte Carlo

Účelem tohoto experimentu bylo ukázat, že algoritmus Monte Carlo dokáže hrát deskovou hru THUD! Za tímto účelem byl proveden experiment ve kterém byl proti nejjednodušší verzi algoritmu Monte Carlo s časem pro výpočet nastaveným na 1 sekundu postaven náhodný hráč v 500 testovacích her. Výsledky tohoto experimentu můžete vidět v tabulce 4.

Z výsledků můžeme usuzovat to, že algoritmus Monte Carlo je i ve své nejjednodušší podobě schopen hrát hru THUD! s nějakou strategií. Je rovněž nutné zmínit, že výsledky tohoto experimentu nemají prakticky žádnou vypovídací hodnotu co se síly algoritmu týče. Jednak proto, že v deskové hře THUD! je jakákoli strategie lepší, než zcela náhodně volit tahy a jednak proto, že algoritmus Monte Carlo měl v tomto experimentu na rozhodnutí jednoho tahu značně krátký časový úsek. Už při prodloužení času výpočtu na 5 sekund můžeme vidět můžeme vidět, že algoritmus všechny hry proti náhodnému hráči zcela ovládl. Viz tabulka experimentu 100 utkání jednoduchého Monte Carla s dobou výpočtu 5 sekund proti náhodnému hráči. Viz tabulka 5.

SimLenCut Monte Carlo (1s, 120 ply) vs. Simple Monte Carlo (1s)	
Výher	41.6%
Proher	30.4%
Remíz	28.0%

Tabulka 6. Výsledky experimentu funkčnosti vylepšení zkrácením simulací algoritmu Monte Carlo.

SimLenCut Monte Carlo (1s, 40 ply) vs. SimLenCut Monte Carlo (1s, 120 ply)	
Výher	87.6%
Proher	11.6%
Remíz	0.8%

Tabulka 7. Výsledky experimentu délky zkrácení simulací algoritmu Monte Carlo.

5.2. Experiment - Vylepšení algoritmu Monte Carlo zkrácením simulací

Předchozí experiment ukázal, že algoritmus Monte Carlo dokáže hrát hru THUD! Bohužel, vzhledem k jeho performanci při výpočtu délky 1 sekundy proti náhodnému hráči můžeme odhadovat, že výkon algoritmu nebude zcela optimální. Proto budu postupně na algoritmus aplikovat vylepšení navržená v kapitole 4. a zkoumat jeho výkon. Jako porovnávací algoritmus budu vždy používat ten, který se v posledním experimentu ukázal jako nejsilnější, v tomto případě tedy základní verzi algoritmu Monte Carlo.

V experimentu byly proti sobě postaveny v 250 testovacích utkání algoritmus Monte Carlo se zkrácenými simulacemi, s výpočetní dobou 1 sekunda a simulacemi zkrácenými na maximálně 120 tahů proti základní verzi algoritmu Monte Carlo s výpočetním časem rovněž na 1 sekundě. Výsledky můžete vidět v tabulce 6.

Z výsledků experimentu vyplývá, že zkrácení simulací skutečně zvýší sílu algoritmu, nicméně by se mohlo zdát, že zvýšení síly není nijak výrazné, takřka 30% remíz a ještě o něco málo vyšší percentil proher napovídají, že síla obou algoritmů si není nijak výrazně vzdálena. To je způsobeno nastavením zkrácení simulací na 120 tahů, což je poměrně vysoká hodnota a to i vzhledem k relativně vysoké průměrné délce hry. (viz. kapitola 3.3.) V dalších experimentech byl proto zkoumán vliv poměrného zkrácení simulací vzhledem k síle algoritmu.

V dalších experimentech byl proti sobě proto postaven Algoritmus Monte Carlo se zkrácením simulací na 40 tahů proti algoritmu Monte Carlo se zkrácením simulací na 120 tahů, Algoritmus Monte Carlo se zkrácením simulací na 20 tahů proti algoritmu Monte Carlo se zkrácením simulací na 40 tahů a konečně Algoritmus Monte Carlo se zkrácením simulací na 5 tahů proti algoritmu Monte Carlo se zkrácením simulací na 20 tahů. Výsledky těchto experimentů můžete vidět v tabulkách 7., 8., 9.

SimLenCut Monte Carlo (1s, 20 ply) vs. SimLenCut Monte Carlo (1s, 40 ply)	
Výher	93.6%
Proher	0.2%
Remíz	4.0%

Tabulka 8. Výsledky experimentu délky zkrácení simulací algoritmu Monte Carlo.

SimLenCut Monte Carlo (1s, 5 ply) vs. SimLenCut Monte Carlo (1s, 20 ply)	
Výher	98.4%
Proher	0.0%
Remíz	1.6%

Tabulka 9. Výsledky experimentu délky zkrácení simulací algoritmu Monte Carlo.

Zkrácení tahů	Průměrně simulací	Medián simulací	Směrodatná odchylka simulací
Bez zkrácení simulací	845.1651	427	913.7661
120 ply	667.0894	503	468.2712
40 ply	1764.29	1689	1029.324
20 ply	3452.877	2740	2358.305
5 ply	8803.382	7061	5193.747

Tabulka 10. Statistické hodnoty počtů simulací u jednotlivých hráčů během experimentu

Jak můžete vidět zkrácení simulací značně zvýšilo výkon algoritmu v hře sám proti sobě. Důvod je zřejmý, zkrácené simulace zaberou pouze zlomek času oproti úplné simulaci, která je navíc kvůli pravidlům hry THUD! o ukončování jednoduché hry problematická a tak dokáže algoritmus provést několikrát více simulací. (Průměrný počet simulací za 1 sekundu klasického přístupu a zkrácených simulací můžete vidět v tabulce 10.) Na druhou stranu, ale algoritmus se zkrácením simulací zároveň ztrácí možnost "vidět dopředu", tedy schopnost plánovat dlouhodobé strategie.

5.3. Experiment - Vylepšení změnou simulační techniky

Zkrácení simulací se v kapitole 5.2. ukázalo jako účinný způsob jak zvýšit hrubou sílu algoritmu Monte Carlo. Zkracování simulací má ale své hranice, kdy nárůst počtu simulací nedokáže vykompenzovat sníženou hloubku prohledávání. Je tedy nutné hledat další cesty jak vylepšit chování algoritmu. V této části se budu zabývat vlivem změny simulačních metod (rozhodovacích AI) na výkon algoritmu.

První provedený experiment byla záměna zcela náhodných simulačních AI za algoritmus Minimax hloubky 1. Ve 250 testovacích hrách byly postupně proti sobě postaveny zcela náhodné simulace a simulační metody kde byly jednotliví hráči v simulacích nahrazeni, pokaždé s doposud nejsilnější metodou. Všechny testovaní hráči měli simulace zkrácené na 20 tahů. Výsledky můžete vidět v tabulce 11.

Můžeme vidět, že zatímco nahrazení trpasličí simulační AI za Minimax hloubky 1 bylo vzhledem k původní verzi kontraproduktivní, tak náhrada simulační AI trpaslíků (tedy strany s řádově větší počtem možných tahů) naopak algoritmus mírně posílilo. Množství výher je takřka vyrovnané, mírně se kloní ve prospěch nahrazené

Hra (Trpasličí simAI - Trollí simAI)	Výher hráč 1	remíz	Výher hráč 2
Random - Random vs. Random - Minimax 1	64.8%	3.6%	31.6%
Random - Random vs. Minimax 1 - Random	28.0%	40.4%	31.6%
Minimax 1 - Random vs. Minimax 1 - Minimax 1	0%	87.6%	12.4%

Tabulka 11. Výsledky experimentu nahrazení náhodných simulačních hráčů Minimaxem hloubky 1

Hra (Trpasličí simAI - Trollí simAI)	Výher hráč 1	remíz	Výher hráč 2
Minimax 1 - Minimax 1 vs. Random - Minimax 2	100%	0%	0%
Minimax 1 - Minimax 1 vs. Minimax 2 - Random	96.8%	3.2%	0%
Minimax 1 - Minimax 1 vs. Minimax 2 - Minimax 2	98.8%	0%	1.2%

Tabulka 12. Výsledky experimentu nahrazení náhodných simulačních hráčů Minimaxem hloubky 2

Typ simulace (Trpasličí AI - Trollí AI)	Průměr	Medián	Směrodatná odchylka
Random - Random	3213.291	2649	2123.053
Random - Minimax1	1754.149	1572	1042.693
Minimax1 - Random	342.343	193	371.0049
Minimax1 - Minimax1	328.9429	254	221.5472
Random - Minimax2	49.69263	12	75.67547
Minimax2 - Random	6.101377	3	8.220827
Minimax2 - Minimax2	4.469744	3	5.45859

Tabulka 13. Statistiky počtu simulací během výpočtu jednoho tahu u jednotlivých simulačních metod

simulační AI a výrazně se zvýšil poměr remíz. U posledního z testovaných hráčů jsem nahradil obě simulační AI. Opět se značně zvýšil počet remíz na úkor výher jednotlivých hráčů, nicméně přestože množství výher se snížilo u nového hráče pouze na 12.4% neprohrál v experimentu ani jednou. Proto budeme tohoto hráče považovat za prozatím nejsilnější získanou verzi algoritmu Monte Carlo.

Viděli jsme, že přechod ze zcela náhodných simulací na složitější rozhodovací algoritmy zvýšila sílu algoritmu. Nyní přirozeně vyvstává otázka, zda by mohly ještě silnější rozhodovací algoritmy dále zvýšit sílu algoritmu Monte Carlo. Pro ověření této hypotézy byl proveden experiment kde byl doposud nejsilnější verze algoritmu postavena proti Monte Carlu, u nějž byly simulační algoritmy nahrazeny postupně algoritmem minimax hloubky 2. Všichni počítačový hráči měli simulace zkráceny na 20 tahů a pro výpočet jim byla dána 1 sekunda. Výsledky můžete vidět v tabulce 12.

Z výsledku je zřejmé, že náhrada simulačních AI za Minimax hloubky 2 byla vzhledem k síle algoritmu kontraproduktivní. Je to zřejmě z toho důvodu, že přestože je algoritmus Minimax do hloubky 2 silnější než jeho varianta do hloubky 1 je rovněž podstatně výpočetně náročnější. Jeho časová složitost roste s hloubkou exponenciálně a tak zatímco algoritmus minimax hloubky 1 s řádovou složitostí $O(n)$ zvýšil přesnost simulací a zároveň dovolil dostatečné množství simulací pro dobrou statistiku, tak jeho varianta hloubky 2 s řádovou složitostí $O()$ výrazně snížil počet simulací a tím i sílu algoritmu Monte Carlo. Toto ilustruje tabulka 13.

5.4. Experiment - síla algoritmu Monte Carlo

V kapitolách 5.1., 5.2. a 5.3. jsem ukázal, že algoritmus Monte Carlo dokáže hrát hru THUD! a použitím jednotlivých navržených vylepšení a jejich testováním jsem postupně vylepšoval jeho výkonnost. V této kapitole budu porovnávat takto odladěný algoritmus s klasickým přístupem reprezentovaným algoritmem Minimax spuštěným do hloubky 2. Pro toto porovnání byl proveden experiment, kde proti sobě hrály algoritmus Monte Carlo, se simulacemi zkrácenými na 5 tahů, s oběma simulačními AI nastavenými jako Minimax hloubky 1 s délkou výpočtu nastavenou na 60 sekund a algoritmus Minimax hloubky 2 v 50 testovacích hrách. Výsledky

PseudoRandomSimLenCut Monte Carlo vs. Minimax 2	
Výher	6.0%
Remíz	16.0%
Proher	78.0%

Tabulka 14. Maximálně vylepšené Monte Carlo vs. Minimax hloubky 2

Simple MCTS (1s) vs. Random AI	
Výher	98.4%
Proher	0.8%
Remíz	0.8%

Tabulka 15. Výsledky experimentu funkčnosti algoritmu MCTS.

můžete vidět v tabulce 14.

Výsledky ukazují, že v této konkrétní implementaci poráží naprosto jasně algoritmus Minimax algoritmus Monte Carlo. Kromě toho má navíc Minimax hloubky 2 podstatně kratší dobu výpočtu (na testovacím stroji obvykle méně než 1 sekundu) než algoritmus Monte Carlo, jež na své rozhodnutí vždy spotřeboval plných 60 sekund (dle svého nastavení). Převaha Minimaxu může být způsobena několika faktory. Například strukturou herního stromu, ten ač značně velký (viz. 3.3.) se sestává ve skutečnosti ze dvou druhů pater, z trpasličího patra, které může v sobě mít i přes 600 uzlů a málokdy mívá méně než 100 uzlů a trollího patra které naopak málokdy přesáhne svou velikostí 64 uzlů. Minimax proto při svém průchodu tímto stromem do hloubky 2 vždy projde jedno trpasličí a jedno trollí patro, takže se rozhoduje podstatně rychleji, než kdyby byla všechna patra stejně velká. Naproti tomu algoritmus Monte Carlo kvůli simulacemi často prochází velká trpasličí patra, která snižují relevanci výsledku simulace. Pseudonáhodné simulace tento problém částečně řeší, ale greedy přístup simulací Minimax 1 může snadno zcela zavrhnout ideální větev.

Dalším důvodem proč algoritmus Monte Carlo neuspěl může být samotná podstata hry THUD! Obzvláště při hře za trpaslíky je v ní nutné používat dlouhodobou taktiku, obětovat figury a stavět se do dobře chráněných obraných bloků. Dlouhodobá strategie by měla být pro algoritmus Monte Carlo výhodnou, problém ale spočívá opět v příliš širokém herním stromu a tudíž nepřesnosti ohodnocení simulací. Navíc algoritmus Minimax v hloubce 2 postupuje takřka "hladovým" přístupem. Zničí veškeré pokusy algoritmu Monte Carlo vybudovat blok hned v počátku.

5.5. Experiment - správnost algoritmu Monte Carlo Tree Search

V předchozích sekcích jsem ukázal, že algoritmus počítačového hráče založený na metodě Monte Carlo dokáže hrát hru THUD! Bohužel ve srovnání s klasickým přístupem Teorie Her zastoupeným algoritmem Minimax se neukázal jako příliš výkonný. V této a dalších sekcích proto provádím experimenty s algoritmem kombinující oba přístupy, algoritmem Monte Carlo Tree Search.

Nejdříve opět ukážeme, že je algoritmus MCTS schopen hrát hru THUD! Za tímto účelem byl proveden experiment, ve kterém byl proti sobě postavena základní implementace algoritmu MCTS na 1 sekundě výpočtu proti zcela náhodnému hráči. Výsledky můžete vidět v tabulce 15..

Z výsledků je zcela jasně vidět, že algoritmus MCTS je schopen hrát hru THUD!. Rovněž si všimněte, že výsledky jsou lepší, než u podobného experimentu s algoritmem Monte Carlo. (viz.5.1.)

Hodnota Konstanty c	Výher	Remíz	Proher
0.1	0%	0%	100%
0.4	0%	0%	100%
0.6	13.3%	6.7%	80%
0.9	0%	0%	100%
1.0	0%	0%	100%
1.5	0%	0%	100%
3.0	6.7%	0%	93.3%

Tabulka 16. První experiment ladění konstanty MCTS

Hodnota Konstanty c	Výher	Remíz	Proher
0.6	93.3%	0%	6.7%
0.65	100%	0%	0%
0.7	100%	0%	0%
0.75	93.3%	0%	6.7%
0.8	80%	13.3%	6.7%
0.85	93.3%	0%	6.7%

Tabulka 17. Druhý experiment ladění konstanty MCTS

5.6. MCTS - Ladění konstanty

V algoritmu MCTS hraje značnou roli tzv. Exploration Exploitation dilema(viz. 2.2.), regulované konstantou c. Laděním hodnoty této konstanty se zabývají následující experimenty.

Je rovněž nutné podotknout, že pro zjednodušení určení konstanty c algoritmus MCTS vnitřně normalizuje ohodnocení desky do intervalu $[0, 1]$. Dále v této podkapitole ukážu proč.

První experiment testoval hodnoty konstanty c v rozsahu 0.1 až 3.0 proti jednomu ze slabších verzí algoritmu Monte Carlo, konkrétně Monte Carlo s pseudonáhodnými simulacemi (Random, Minimax) zkrácenými na 20 tahů. Výsledky můžete vidět v tabulce 16.

Z výsledků je vidět, že nejlépe si algoritmus MCTS počínal při konstantě c nastavené na hodnotu 0.6. Abych toto tvrzení potvrdil provedl jsem další experiment jež podobným způsobem testoval konstanty v rozmezí 0.6 až 0.85, tentokrát byly však postaveny proti zcela základní verzi algoritmu Monte Carlo. Výsledky můžete vidět v tabulce 17.

Zde už můžeme vidět, že konstanta nejlepší výsledky dává algoritmus MCTS při konstantě c nastavené okolo hodnoty 0.7. Pokud uvážíme, že algoritmus vnitřně normalizuje skóre do intervalu $[0, 1]$ nabízí se nám možnost, že ideální hodnota konstanty by se mohla rovnat $\frac{1}{\sqrt{2}}$, jak je tomu podle [3] u mnoha her kde lze hodnotu desky vyjádřit v intervalu $[0, 1]$. Abych toto tvrzení potvrdil provedl jsem experiment, kde byl algoritmus MCTS s konstantou c nastavenou na hodnotě $\frac{1}{\sqrt{2}}$ postaven proti oběma z předchozího experimentu nejsilnějším nastavením, tedy MCTS s konstantou c na hodnotách 0.65 a 0.7. Výsledky můžete vidět v tabulce 18.

Jak můžeme z výsledků experimentu vidět, hodnota $\frac{1}{\sqrt{2}}$ se neukázala jako vhodná pro konstantu c algoritmu MCTS. Naproti tomu se zdá, že algoritmus s $c = 0.65$ nám dává nejlepší výsledky. Proto bude v dalších experimentech s algoritmem MCTS použita konstanta c v hodnotě 0.65.

Hra	Výher hráč 1	Remíz	Výher hráč 2
MCTS($c = 0.65$) vs. MCTS($c = \frac{1}{\sqrt{2}}$)	50.0%	15.2%	34.8%
MCTS($c = 0.7$) vs. MCTS($c = \frac{1}{\sqrt{2}}$)	43.2%	13.2%	43.6%

Tabulka 18. Třetí experiment ladění konstanty MCTS

Hra c	Výher hráč 1	Remíz	Výher hráč 2
SimLenCut MCTS(120 ply) vs Simple MCTS	35.6%	21.2%	43.2%
SimLenCut MCTS(20 ply) vs Simple MCTS	32.4%	22.8%	38.8%
SimLenCut MCTS(5 ply) vs Simple MCTS	40.4%	20.0%	39.6%

Tabulka 19. Experiment zkrácení simulací algoritmu MCTS

5.7. Experiment - Vylepšení MCTS zkrácením simulací

V předchozí podkapitole jsem odladil exploration - exploitation konstantu pro algoritmus MCTS, tím jsem získal nejlépe pracující základní verzi algoritmu. V této a dalších podkapitole se budu zabývat dalším vylepšením algoritmu MCTS.

Stejně jako u algoritmu Monte Carlo se zde nabízí vylepšení využitím zkrácených simulací. To by mělo mít za následek zvýšení počtu vzorků odebraných algoritmem (v případě MCTS tedy vznikne větší část herního stromu) a tím i přesnost vybraného tahu. Za účelem otestování efektu zkrácení simulací na algoritmus MCTS byl proveden experiment, kde byl postupně proti sobě vždy ve 250 hrách postaven základní algoritmus MCTS a jeho verze s různou měrou zkrácenými simulacemi, všechny spuštěny na 1 sekundě výpočtu na tah. Výsledky můžete vidět v tabulce 19.

Jak je vidět z výsledků experimentu, ani značné zkrácení simulací na 5 tahů výrazně nezvýšilo sílu algoritmu, naopak menší zkrácení simulací se ukázalo jako kontraproduktivní, protože algoritmy s délkou simulací 120 a 20 tahů si počínaly mírně hůř než algoritmus bez zkrácených simulací.

Ještě větší zkrácení simulací by pouze snížilo váhu ohodnocení jedné simulace a nezvýšilo by o mnoho počet simulací, navíc z experimentu vyplývá, že výrazně vyšší počet simulací nezvyšuje sílu algoritmu tak jak to bylo u algoritmu Monte Carlo. Za nejsilnější verzi algoritmu budu tedy dále považovat verzi bez zkrácení simulací

5.8. Experiment - Vylepšení MCTS změnou simulační techniky

Zkrácení simulací se u algoritmu MCTS ukázalo na rozdíl od algoritmu Monte Carlo jako neefektivní vylepšení. Nabízí se tedy vyzkoušet aplikovat na algoritmus i druhé vylepšení algoritmu Monte Carlo a to změnu simulační techniky. Za tímto účelem byl proveden experiment kde byly proti sobě vždy v 250 hrách proti sobě postaveny počítačoví hráči MCTS s jednotlivými simulačními AI nahrazenými algoritmem Minimax hloubky 1 proti doposud nejsilnější získané verzi algoritmu MCTS. Výsledky můžete vidět v tabulce 20.

Z výsledků experimentu můžeme usuzovat, že záměna simulačních AI za Minimax hloubky 1 zvýšila sílu algoritmu MCTS. Zejména záměna trollí simulační AI výrazně přispěla k síle algoritmu, který takto upravený dokázal vyhrát nad svou neupravenou verzí v 82.8% her v experimentu. Naopak a na rozdíl od algoritmu Monte Carlo, výměna pouze simulační AI trpaslíků se ukázala jako neefektivní. Výměna obou simulačních AI za minimax 1 potom přinesla pouze mírné, nicméně stále významné zlepšení.

Hra (Trpasličí simAI - Trollí simAI)	Výher hráč 1	remíz	Výher hráč 2
Random - Random vs. Random - Minimax 1	8.4%	8.8%	82.8%
Random - Minimax 1 vs. Minimax 1 - Random	100%	0%	0%
Random - Minimax 1 vs. Minimax 1 - Minimax 1	41.6%	11.2%	47.2%

Tabulka 20. Výsledky experimentu nahrazení náhodných simulačních hráčů Minimaxem hloubky 1

Hra (Trpasličí simAI - Trollí simAI)	Výher hráč 1	remíz	Výher hráč 2
Random - Minimax 2 vs. Minimax 1 - Minimax 1	0.4%	0.4%	99.2%
Minimax 2 - Random vs. Minimax 1 - Minimax 1	0%	0.4%	99.6%
Minimax 2 - Minimax 2 vs. Minimax 1 - Minimax 1	100%	0%	0%

Tabulka 21. Výsledky experimentu nahrazení náhodných simulačních hráčů Minimaxem hloubky 2

Jak se zdá tak se algoritmus MCTS vzhledem k výměně simulačních AI chová (až na drobné výjimky) podobně jako algoritmus Monte Carlo. (viz. 5.3.) Očekávaný výsledek nasazení Minimaxu hloubky 2 jako simulační AI je proto snížení síly algoritmu. Abych toto potvrdil, byl proveden následující experiment: Vždy ve 250 hrách byly proti sobě postaveny počítačoví hráči MCTS s jednotlivými simulačními AI nahrazenými algoritmem Minimax hloubky 2 proti doposud nejsilnější získané verzi algoritmu MCTS. Výsledky můžete vidět v tabulce 21.

Jak je z výsledků experimentu vidět, předpoklad se potvrdil. Nahrazení simulačních AI Minimaxem hloubky 2 vedlo k výraznému snížení počtu odebraných vzorků (MCTS tedy vybudovalo pouze menší část herního stromu) což výrazně snížilo sílu algoritmu.

5.9. Experiment - Síla algoritmu MCTS

V předchozích podkapitolách jsem testoval různé verze algoritmu MCTS s různým nastavením abych tak získal co nejsilnější algoritmus. Nyní takto získaný algoritmus otestuji proti nejsilnější získané verzi algoritmu Monte Carlo (viz. 5.3.) a proti klasickému přístupu reprezentovanému algoritmem Minimax.

Nejdříve byl tedy proveden experiment kde byla nejsilnější verze algoritmu MCTS (MCTS bez zkrácení simulací, s konstantou $c = 0.65$, se simulačními technikami Minimax hloubky 1 za oba hráče) proti nejsilnější verzi algoritmu Monte Carlo (Simulace zkrácené na 5 tahů, Obě simulační AI jsou Minimax hloubky 1), oba algoritmy měly na výpočet tahu 1 sekundu a bylo spuštěno 250 her. Výsledky tohoto experimentu můžete vidět v tabulce 22.

Z experimentu je zřejmý absolutní neúspěch algoritmu MCTS. Algoritmus MCTS, ač se zdál v základní verzi této implementace mírně silnější než algorit-

PseudoRandomMCTS vs. PseudoRandom SimLenCut Monte Carlo	
Výher	0.0%
Proher	100.0%
Remíz	0.0%

Tabulka 22. Výsledky experimentu síly algoritmu MCTS, proti algoritmu Monte Carlo.

PseudoRandomMCTS vs. Minimax 2	
Výher	0.0%
Proher	100.0%
Remíz	0.0%

Tabulka 23. Výsledky experimentu síly algoritmu MCTS, proti algoritmu Minimax.

mus založený na metodě Monte Carlo, se ukazuje být v odladěných verzích této implementace slabší, protože navrhovaná vylepšení byla buďto kontraproduktivní, nebo sílu algoritmu zvýšila jen mírně. Algoritmus MCTS se tedy v této implementaci ukázal jako slabší, než algoritmus založený na metodě Monte Carlo.

Dále byl proveden experiment pro porovnání síly algoritmu MCTS v této implementaci hry THUD! s klasickým přístupem k počítačovému hráči založeném na teorii her. Za tímto účelem byl proveden experiment kde byly proti sobě v 50 hrách postaveny Nejsilnější verze algoritmu MCTS a Minimax s hloubkou 2. Výsledky můžete vidět v tabulce 23.

Vzhledem k výsledkům předchozího experimentu nebyl výsledek příliš překvapivý. V podkapitole 5.4. porazil algoritmus Minimax hloubky 2 algoritmus založený na metodě Monte Carlo, v předchozím experimentu porazil tentýž algoritmus založený na metodě Monte Carlo tuto implementaci algoritmu MCTS, nyní tedy Minimax hloubky 2 porazil v experimentu algoritmus MCTS a v této implementaci se tedy ukázal jako nejsilnější algoritmus počítačového hráče hry THUD!

6. Závěr

Představil jsem deskovou hru THUD! a její analýzou jsem postupně získal její stavovou složitost ($4,08 \cdot 10^{90}$) a horní odhad složitosti herního stromu (10^{423}). Vytvořil jsem implementaci pravidel této hry a v jejím rámci několik druhů počítačových hráčů, zejména počítačové hráče založené na metodě Monte Carlo a algoritmu Monte Carlo Tree Search. Tyto počítačové hráče jsem s pomocí experimentů postupně vylepšoval a nakonec srovnával s klasickým přístupem k počítačovým Hráčům v deskových hrách, reprezentovaných algoritmem Minimax. V mé implementaci hry THUD! se přístup metodou Monte Carlo oproti klasickému přístupu neosvědčil a algoritmus Monte Carlo Tree Search, vykazoval ještě horší výsledky než přístup Monte Carlo.

Reference

[1] Monte-Carlo Search Techniques in the Modern Board Game Thurn and Taxis

[2] Bandit Based Monte Carlo Planning

[3] A Survey of Monte Carlo Tree Search Methods Cameron Browne, Member, IEEE, Edward Powley, Member, IEEE, Daniel Whitehouse, Member, IEEE, Simon Lucas, Senior Member, IEEE, Peter I. Cowling, Member, IEEE, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis and Simon Colton

[4] Random number generation and Monte Carlo methods / James E. Gentle. New York, N.Y. : Springer, c2003 xv, 381 s.

[5] Georges-Louis Leclerc, Comte de Buffon, Lenka Štefková Bakalářská práce

[6] The begining of the Monte Carlo method, J. Metropolis

[7] Progressive strategies for Monte-Carlo tree search, guillaume M.J-B. Chaslot, Mark H.M. Winands,

[8] www.thudgame.com, The official THUD! website.

[9] H. J. van den Herik, J. W. H. M. Uiterwijk, and J. van Rijswijck. Games solved: now and in the future. Artificial Intelligence, 134(1-2), 2002.

[10] M. P. D. Schadd, M. H. M. Winands, J. W. H. M. Uiterwijk, H. J. van den Herik, and M. H. J. Bergsma. Best Play in Fanorona leads to Draw. New Mathematics and Natural Computation 4, (3):369–387, 2008.

[11] J.A.M. Nijssen. Using intelligent search techniques to play the game Khet. Master thesis, Maastricht University, 2009.

[12] Yoshikuni Sato, Daisuke Takahashi, Reijer Grimbergen A shogi program based on Monte-Carlo Tree Search, Tsukuba and Tokyo, Japan

[13] Ye Fan, Bandit Algorithms in Game Tree Search: Application to Computer Renju, Department of Computer Science University of British Columbia Vancouver