Univerzita Karlova v Praze

Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE

Jakub Tomek

Aplikace MCTS na hru Quoridor

Katedra teoretické informatiky a matematické logiky

Vedoucí bakalářské práce: RNDr. Jan Hric

Studijní program: informatika

Studijní obor: obecná informatika

Praha 2011

Děkuji svému vedoucímu RNDr. Janu Hricovi za to, že mě přivedl k Monte Carlo metodám umělé inteligence, za pomoc při jejich zkoumání a analýze a za pomoc s bakalářkou prací.

Dále děkuji knihovně Matematicko-fyzikální fakulty za zapůjčení odborné literatury.

Nakonec děkuji programu MSDN Academic Alliance za poskytnutí softwarových vývojových nástrojů, které jsem pro práci použil.

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.
Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona.

V ……..…..…..……..... dne…..... podpis

Název práce: Aplikace MCTS na hru Quoridor

Autor: Jakub Tomek

Katedra / Ústav: Katedra teoretické informatiky a matematické logiky

Vedoucí bakalářské práce: RNDr. Jan Hric

Abstrakt: Monte Carlo Tree Search je relativně novou technikou prohledávání stromu navrženou pro počítačového hráče v hrách, které mají příliš velký stavový prostor na to, aby šel efektivně prohledávat deterministickým algoritmem. MCTS v základní verzi poskytuje jednoduchý způsob ohodnocování pozic bez jakýchkoliv doménově specifických znalostí. MCTS byl již aplikován v mnoha variantách pro počítačové Go, jeho použití na ostatní hry však dosud není zdaleka tak hluboce prozkoumáno. Tato práce se zabývá možností použití MCTS na jednu konkrétní hru, a to Quridor.

Klíčová slova: Monte Carlo Tree Search, Quridor, UCT

Title: Application of MCTS on game Quoridor

Author: Jakub Tomek

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: RNDr. Jan Hric

Abstract: Monte Carlo Tree Search is quite a new technique for searching a tree developed for a computer player in games, that have too large state space to be effectively searched by an deterministic algorithm. MCTS in its basic version offers a simple way to evaluate positions without any domain specific knowledge. MCTS was already applied in many variants for computer Go, however its usage for other games has not been nearly as deep studied. This work deals with the option of using MCTS on a particular game called Quoridor.

Keywords: Monte Carlo Tree Search, Quoridor, UCT

Obsah

Úvod

Monte Carlo metody umělé inteligence se používají jako možná alternativa k deterministickým algoritmům, které se při řešení některých velmi složitých problémů co do efektivity dosud nedokážou vyrovnat lidskému myšlení. V roce 2006 se zvedla vlna zájmu o Monte Carlo algoritmy poté, co program CrazyStone využívající Monte Carlo Tree Search vyhrál Počítačovou olympiádu v Turíně pořádanou asociací ICGA (International Computer Game Association) ve hře Go na hrací desce 9x9 a porazil dosud nejsilnější program Go Intellect. Brzy poté se začaly objevovat nové programy s umělou inteligencí inspirované metodou Monte Carlo Tree Search a různá rozšíření této metody, ze kterých je nejvýznamnější a nejúspěšnější rozšíření UCT (Upper Confidence Bounds applied for Trees). Program MoGo, založený na metodě MCTS s rozšířením UCT vyhrál Počítačovou olympiádu v roce 2007 v Amsterdamu v Go19x19, nechaje za sebou i původní CrazyStone.

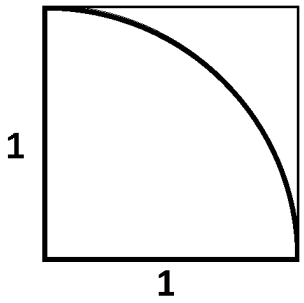
Monte Carlo Tree Search v sobě nese odpověď na problém příliš velkého stavového prostoru, který není možné efektivně prohledávat deterministickým algoritmem. MCTS neprochází všechny možné alternativy, ale porovnává výsledky náhodných průchodů stavovým prostorem (stromem hry) a na základě výsledků těchto náhodných průchodů pak volí další postup. Díky tomu dokáže prohledat do velké hloubky i rozsáhlý strom bez ohledu na jeho větvící faktor. Další výhodou MCTS je, že ve své čisté podobě nepotřebuje žádné doménově závislé znalosti o úloze, kromě rozeznání úspěchu a neúspěchu. Obecně je umělá inteligence založená na metodě Monte Carlo strategicky silným algoritmem, těžícím z kvalitní statistiky ale takticky slabým, protože možné vynechání některé z možností při testování může mít fatální vliv na vyhodnocení situace. V konkrétních aplikacích MCTS je vždy snaha eliminovat tuto slabinu a maximalizovat efektivitu algoritmu pomocí různých heuristik a často doménově závislých znalostí, které byly úspěšně použity například v programu MoGo.

Tato práce se zabývá možností využití Monte Carlo tree Search na hru Quoridor, efektivitou algoritmu pro danou úlohu a možností rozšíření a heuristik pro optimalizaci algoritmu, jejichž účinnost ověřuje testováním na programu, který je přílohou práce, v testovacím módu (MCTS-Quoridor, [1]).

1. Monte Carlo metody a jejich dosavadní využití

1.1. Co je Monte Carlo?

Monte Carlo je označení pro širokou rodinu matematických metod, které pro řešení konkrétní ch úloh využívají náhodné jevy, které jsou pak konfrontovány s teorií pravděpodobnosti. Typickým názorným příkladem použití Monte Carlo metody je výpočet čísla π pouze s použitím generátoru náhodných čísel. Uvažujme následující obrazec:

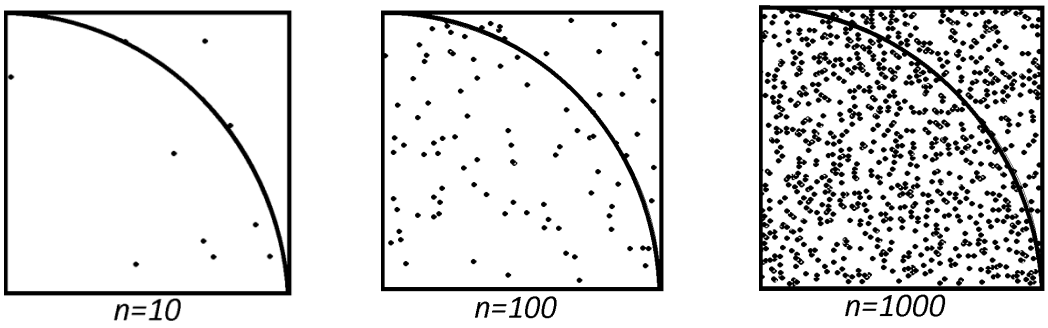


Obr. 1.1

Čtvrtkruh vepsaný do čtverce o hraně délky 1 má obsah $\frac{\pi}{4}$. Z teorie pravděpodobnosti víme, že zvolíme-li náhodný bod uvnitř čtverce, pak pravděpodobnost *P*, že tento bod bude ležet uvnitř čtvrtkruhu je rovna poměru obsahu čtvrtkruhu a obsahu celého čtverce, tedy

$$P = \frac{\frac{\pi}{4}}{1} = \frac{\pi}{4}$$

Jestliže nyní zvolíme určitý počet náhodných bodů uvnitř čtverce, bude podíl bodů ležících uvnitř čtvrtkruhu přibližně odpovídat pravděpodobnosti *P*. Čím více bodů použijeme, tím více se jí bude blížit.



Obr.1.2: Aproximace π

Velmi názorná je pro tento příklad statistická (frekvencionistická) definice pravděpodobnosti:

$$P = \lim_{n\to\infty} \frac{k}{n}$$

Kde *k* je počet pozorovaných jevů (bod leží uvnitř čtvrtkruhu) a *n* je počet pokusů. Předpokládáme-li, že geometrická a statistická definice pravděpodobnosti jsou v tomto případě ekvivalentní, pak vychází zřejmá rovnost

$$\lim_{n\to\infty} \frac{k}{n} = \frac{\pi}{4}$$

která dokazuje smysluplnost této metody výpočtu π. Implementace této metody jako počítačového programu je triviální. K rozlišení toho, zda se bod nachází uvnitř čtvrtkruhu lze použít Pythagorovu větu.

1.2. Vznik Monte Carlo metod a jejich využití mimo UI

Pravděpodobně nejstarší zdokumentovaný experiment založený na metodě náhodných pokusů popsal v roce 1777 francouzský matematik Comte de Buffon. (Kalos, Whitlock, [2]) V experimentu je jehla délky *L* náhodně vhazována na vodorovnou desku s vyznačenými rovnoběžnými liniemi, které jsou od sebe vzdáleny konstantní délku *d*. Množstvím opakovaných hodů se de Buffon pokouší odhadnout pravděpodobnost *P*, že jehla dopadne na jednu z linií na desce. Buffon zároveň pomocí integrální geometrie vypočítal, že

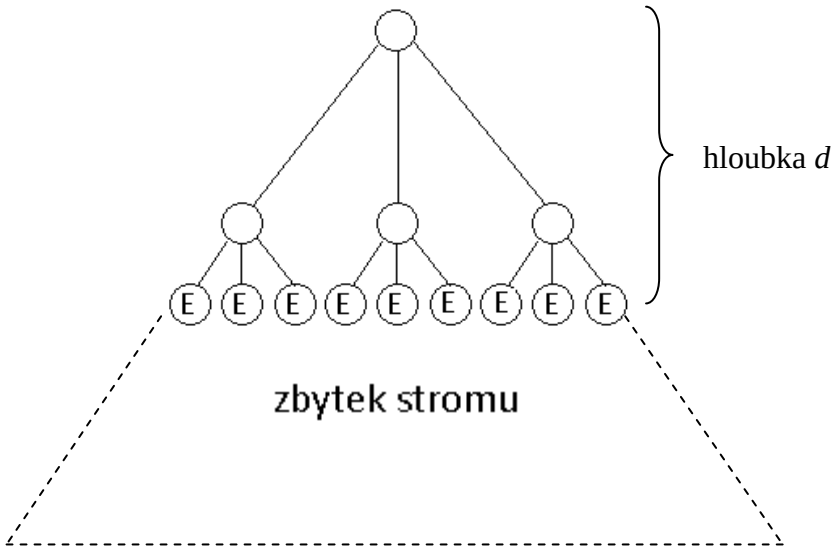
$$P = \frac{2L}{\pi d}$$

O několik let později přišel francouzský matematik Pierre Simon de Laplace s návrhem využít tohoto experimentu k výpočtu čísla π. Tento postup je velmi podobný výpočtu π popsanému na začátku kapitoly. Úloha je dnes všeobecně známá pod názvem "Buffonova jehla".

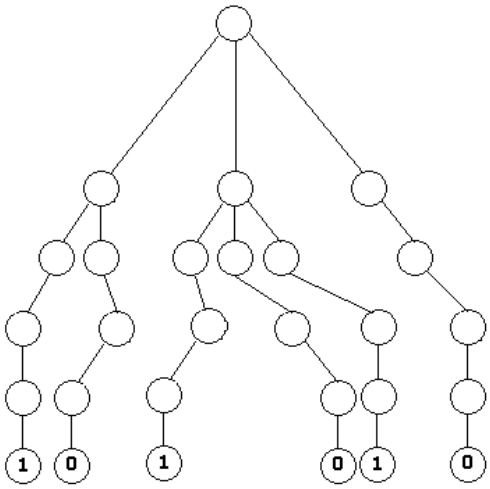
Pojem Monte Carlo ve smyslu matematické metody byl poprvé použit ve 40. letech 20. století vědci v Los Alamos, kteří pracovali na vývoji jaderných zbraní. Základem metody je provedení velkého množství náhodných pokusů nebo simulací, jejichž výsledky je možné použít ke studiu nějaké řešené úlohy. Ačkoliv není metoda Monte Carlo nutně vázána na počítače, teprve vývoj moderních počítačů ve druhé světové válce způsobil přelom pro tuto metodu a možnost rychlého provádění mnoha opakovaných simulací z ní učinila účinný nástroj pro řešení mnoha úloh, zejména výpočet určitých integrálů. Na počátku 50. let 20. století se zvedla vlna zájmu o Monte Carlo metody a byly vytvořeny teoretické postupy pro využití Monte Carla k řešení úloh ze statistické mechaniky, šíření záření, ekonomického modelování a dalších oblastí. Výkon tehdejších počítačů však v mnoha případech neumožňoval uvést teoretické postupy do praxe a tak byly mnohé metody použity až o mnoho let později díky vývoji počítačů a současně i díky vývoji a zlepšování samotných metod.

1.3. Monte Carlo Tree Search

Monte Carlo Tree Search je metoda umělé inteligence původně navržená pro počítačové Go. Go je pravděpodobně nejsložitější všeobecně rozšířená desková hra na světě. Její složitost je dána velkým množstvím možných pozic, kterých je řádově 10^{171}. Jak již bylo zmíněno v úvodu, MCTS přináší možné řešení problému příliš velkého stavového prostoru. Zatímco deterministické algoritmy procházejí všechny větve stromu do dané hloubky, ve které použijí heuristickou ohodnocovací funkci na výslednou pozici hry, MCTS naopak prochází jen některé větve, ale za to projde strom až k listu, tedy do konce hry.

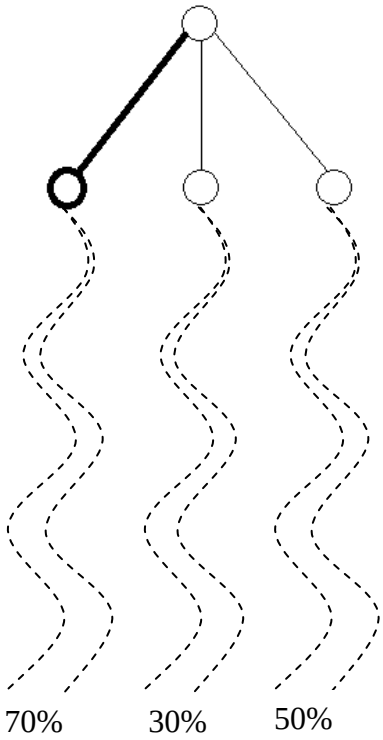


Obr 1.3: Klasický postup deterministického algoritmu: V konstantní hloubce d je na aktuální pozici použita ohodnocovací funkce E.



Obr. 1.4: Monte Carlo algoritmus: Pro každý z možných tahů je provedeno množství náhodných simulací. Vybrané větve se prochází až do listu, ostatní větve jsou vynechány. Každý list udává výsledek hry, ohodnocovací funkce proto není zapotřebí.

Základní způsob výběru dalšího tahu metodou Monte Carlo spočívá ve vybrání tahu s nejlepším poměrem vítězství ku počtu simulací. (Tah, který vede k vítězství s nejvyšší pravděpodobností, je považován za nejlepší)



Obr. 1.5: Výběr tahu s nejvyšší relativní úspěšností

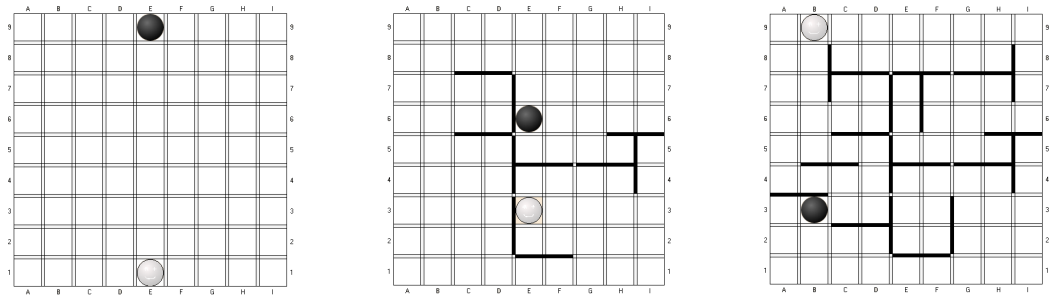
Bylo dokázáno, že algoritmus konverguje v minimax hodnotě pro počet simulací jdoucí k nekonečnu. (Gelly, [5]) Při běžných časech na přemýšlení se typicky podaří provést řádově tisíce až desetitisíce simulací. Kromě základního algoritmu se používají doménově specifické heuristiky na odřezání některých větví stromu nebo rozložení pravděpodobnosti pro průchody jednotlivých větví buď na základě pozice nebo na základě průběžných výsledků simulací.

2. Quoridor

2.1. Pravidla hry Quoridor

Quoridor je desková hra s pro dva hráče. Hraje se na čtvercové desce 9x9 polí, přičemž mezi každými 2 sousedními poli je prostor pro úzkou překážku. Každý hráč má jednu figurku, která na začátku hry stojí uprostřed poslední řady (nejblíže ke svému hráči). Cílem hry je přemístit svojí figurku na protější stranu hrací plochy. Hráči se po tazích střídají. Ve svém tahu může hráč buď přesunout svojí figurku o 1 pole v jednom ze 4 směrů (nahoru, dolů, doleva, doprava) nebo položit na hrací pole překážku. Při pohybu figurkou existuje jedna výjimka, a to možnost přeskočení soupeřovy figurky v případě, že stojí bezprostředně vedle hráčovy a brání jí tím v pohybu jedním směrem. Překážky se pokládají do úzkých mezer mezi poli a mají délku 2 polí. Mezi poli, mezi kterými byla položena překážka, nelze projít s figurkou. Překážka může být položena jen do dvojice mezer, které jsou volné, nesmí křížit jinou překážku a nesmí žádného hráče odříznout takovým způsobem, že by se už nemohl dostat do cíle. Každý hráč má na začátku hry k dispozici 10 překážek.

Možné modifikace pravidel jsou změna velikosti hrací plochy, počáteční polohy figurek, počátečního počtu překážek či modifikace počtu hráčů na 3 nebo čtyři, přičemž každý hráč začíná na jedné straně a jeho cílem je přemístit se na protější stranu. Tato práce se zabývá všemi těmito úpravami kromě změny počtu hráčů a považuje je za volitelné parametry hry.



Obr. 2.1: Začátek, průběh a konec hry

2.2. Aplikace MCTS na Quoridor

Základním parametrem pro algoritmus výběru tahu je současná pozice hry. Algoritmus (respektive program, který ho implementuje) také musí mít přehled o parametrech pro samotnou hru. V každé pozici je možné provést až 132 různých tahů na desce 9x9 polí (4 figurkou + 128 překážkou). V základní variantě MC by algoritmus pro každý z možných tahů provedl určitý počet simulací a poté vybral nejúspěšnějšího potomka kořene za tah k provedení. V práci je však použit mechanismus UCT, který bude podrobně popsán v další kapitole.

Při výběru náhodného tahu, ať už při simulované hře, nebo při hledání potomků uzlu, je potřeba brát ohledy na pravidla týkající se pokládání překážek, což činí jeden ze základních kroků algoritmu podstatně složitějším, než v Go, kde jsou povolené tahy triviálně všechna volná pole. V Quoridoru je triviální pouze pohyb figurkou, ale položení překážky je přípustné pouze pokud položená překážka nekříží jinou a zároveň neznemožňuje žádnému hráči dojít do cíle. Zjištění této skutečnosti samo o sobě má v obecném případě lineární asymptotickou složitost vzhledem k počtu polí, na čtvercové hrací desce tedy kvadratickou vzhledem k délce strany. Při výběru náhodného tahu se proto používá metoda pokus – omyl, tj. zvolení libovolného tahu bez ohledu na pravidla pokládání překážek s následnou kontrolou souladu tohoto tahu s pravidly. Je-li vybraný tah proti pravidlům, je zaznamenán do seznamu zakázaných tahů v paměti a pak se procedura opakuje s tím, že nesmí být vybrán zakázaný tah, dokud není nalezen tah v souladu s pravidly. Díky tomuto postupu není nikdy potřeba vytvářet seznam všech možných tahů v dané pozici, nicméně i tak je výběr náhodného tahu poměrně náročný, zejména v situaci, kdy je hrací plocha hustě obsazena překážkami a podstatná část pozic pro položení další překážky je tím vyloučena. Pro zjednodušení postupu výběru je možné uchovávat seznam zakázaných tahů v paměti po celou hru. Tato optimalizace je implementována v programu a její účinnosti bude věnována pozornost později.

Dalším důležitým prvkem v MCTS je simulovaná hra (tzv. playout), která je základem pro statistiku v algoritmu. Jde o dohrání partie z výchozí pozice až do konce náhodnými tahy. Pro tento postup tedy není potřeba žádné kritérium rozhodování, stačí pouze generátor náhodných čísel. Přesto je možné dosáhnout

zlepšení výsledků umělé inteligence i úpravou tohoto prvku algoritmu, a to jednak přiřazením různých pravděpodobností výběru jednotlivým tahům na základě vhodně zvoleného kritéria (tahy, které se jeví jako lepší, mají vyšší pravděpodobnost výběru) a jednak zkrácením náhodných simulací, které není nutné dohrávat až do konce, ale po určitém počtu tahů je proveden odhad výsledku. Každá z těchto dvou úprav má výhody i nevýhody týkající se kvality odhadu výsledku a složitosti simulace, která je zásadní pro časovou náročnost celého algoritmu, protože jde o časově nejsložitější a mnohokrát opakovaný krok algoritmu. V Quoridoru na rozdíl od jiných her žádné pravidlo nebrání nekonečnému opakování tahů. Zatímco například v Go se hra s každým tahem nevyhnutelně blíží ke konci, protože hrací deska se zaplňuje kameny, v Quoridoru při náhodné simulaci hrozí velmi dlouhá hra způsobená absencí jakékoliv snahy náhodného hráče dostat svou figurku do cíle. Z tohoto důvodu se jeví rozumné použít některou ze zmíněných úprav, například v podobě strategie v playoutu, preferující tahy směřující k cíli. Účinnost těchto úprav bude prověřena v dalších kapitolách.

Protože MCTS se rozhoduje na základě statistiky, tedy se ubírá směrem, který se ve "většině případů" jeví dobře, je možné, že omylem vyhodnotí špatný tah jako dobrý proto, že tento tah vede k prohře jen při konkrétním protitahu soupeře, respektive při protitazích, kterých je ale v celkovém měřítku méně než u jiných vyhodnocovaných tahů. Z tohoto důvodu bývají programy UI (umělé inteligence) založené na MCTS silné strategicky, ale slabé takticky. Přestože MCTS pro svou funkci nevyžaduje žádné doménově specifické informace o kvalitě jednotlivých tahů (viz absence ohodnocovací funkce, kapitola 1.3.) , ukazuje se vhodné přidat do algoritmu doménově specifická pomocná pravidla pro rozhodování, která by kompenzovala obecnou taktickou slabost algoritmu.

Stejně jako u ostatních algoritmů umělé inteligence je i pro MCTS zásadním parametrem ovlivňujícím jeho účinnost čas na přemýšlení. U MCTS můžeme místo času použít jako parametr počet provedených simulací. Zřejmě čím více simulací algoritmus provede, tím přesnější je jeho konečná představa o kvalitě jednotlivých tahů. Jelikož se ale časová náročnost jedné simulace může výrazně lišit při různých parametrech algoritmu, je třeba rozlišovat podle počtu simulací a podle spotřebovaného času.

Quoridor byl poprvé představen v roce 1997, jejím autorem je designér Mirko Marchesi, který se již v roce 1975 podílel na autorství hry podobných charakteristik vydané pod názvem Blockade. Vzhledem k relativně krátké existenci této hry není zatím strategie a taktika prozkoumána tak hluboce, jako u nejrozšířenějších her srovnatelné složitosti. Zatímco některé jednodušší hry již byly vyřešeny úplně, u složitějších se dobře uplatňují deterministické algoritmy a u nejsložitějších (Go) mají z počítačových hráčů zatím nejlepší výsledky programy založené na Monte Carlo metodě. (Coulom, [3])

Tabulka rozšířených deskových her a aplikací programů UI na ně:

Hra	Složitost	Stav
Piškvorky 3x3	10^3	Vyřešeno manuálně
Connect four	10^{14}	Vyřešeno v roce 1988
Dáma 8x8	10^{20}	Vyřešeno v roce 2007
Quoridor 9x9	**10^{20} ***	**?**
Šachy	10^{50}	Programy > lidský hráč
Go	10^{171}	Programy << lidský hráč

Složitostí je v tabulce míněn počet možných pozic ve hře.

Z tabulky je vidět, že ve srovnání s jinými hrami není složitost Quoridoru tak velká, že by s pomocí moderních počítačů nebylo možné hru vyřešit. Na druhou stranu například vyřešení Dámy trvalo 18 let vývoje programu Chinook, který obsahuje objemné knihovny zahájení, koncovek a partií velmistrů a během výpočtů bylo použito až 200 stolních počítačů současně. Přesto se jedná pouze o slabé vyřešení, tzn. program má neprohrávající strategii v počáteční pozici, ale neumí najít optimální tah v libovolné pozici.

* Odhad složitosti: $C = 81^2 \cdot \sum_{i=0}^{20} \frac{\prod_{j=0}^{i} 128-4 \cdot j}{i!}$

3. UCT

3.1. Úvod do UCT

UCT je zkratka pro Upper Confidence Bounds applied for Trees, v češtině horní meze nejistoty aplikované na stromy. Jedná se o významné vylepšení algoritmu MCTS použité poprvé v programu MoGo, který byl představen na počítačové olympiádě v Amsterdamu v roce 2007 týmem francouzských informatiků.

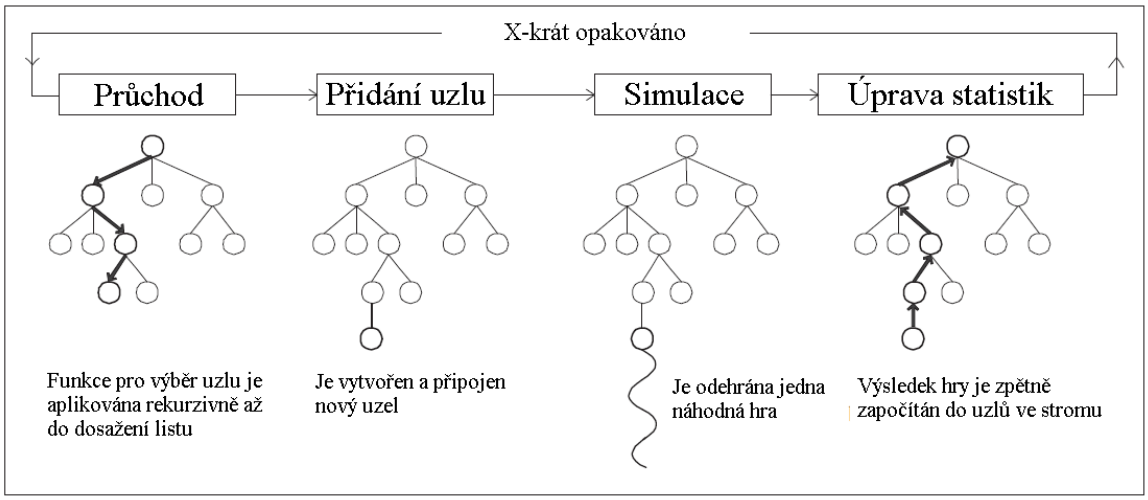
Jednou z možných implementací UCT je cyklický algoritmus, jehož jeden cyklus můžeme rozdělit do čtyř kroků:

1. Průchod stromem od kořene k listu
2. Vytvoření a připojení nového uzlu
3. Dohrání partie náhodnými tahy
4. Úprava statistik stromu

UCT využívá myšlenky, že tahy, které se při dosavadním testování ukázaly jako úspěšnější, by měly být prozkoumávány častěji, než ostatní. Za tímto účelem program staví v paměti částečný strom hry, tvořený uzly reprezentujícími jednotlivé pozice, ve kterých se průběžně ukládají a aktualizují informace o úspěšnosti dané pozice v náhodných simulacích (tzn. o úspěšnosti tahu vedoucího do této pozice) a o počtu průchodů algoritmu skrz uzel. Tento strom je při každé jednotlivé simulaci (playoutu) postupně tvořen přidáváním dalších uzlů a úpravou statistik, typicky dokud nevyprší přidělený čas na přemýšlení nebo není proveden požadovaný počet simulací. Po dokončení simulací je vybrán tah k provedení (potomek kořene) podle heuristického kritéria, typicky uzel s nejvyšším win rate (relativní úspěšnost) nebo uzel s nejvyšším počtem průchodů, přičemž na relativní úspěšnost má v této implementaci pouze počet výher / proher, nebere se ohled na to, o kolik tahů se zvítězilo.

Stavba částečného stromu hry v paměti je cyklický algoritmus, který v každém cyklu přidá do stromu jeden nový uzel (v případě jiných implementací i více). Průchod stromem, který již byl vytvořen v předchozích cyklech algoritmu,

není prováděn náhodně, ale je řešen pomocí kritérií, jejichž parametry jsou win rate a počty průchodů porovnávaných uzlů a jejich rodičů. Tato kritéria budou podrobně rozebrána později. Zbytek partie, tedy tahy, které dosud nejsou součástí stromu v paměti, je dohrán náhodně. Výsledek této partie je zaznamenán do uzlů které se této partie účastnily a její první tah, který není ve stromu, je do něj přidán jako nový uzel.



Obr 3.1: Schéma stavby stromu

3.2. Průchod stromem od kořene k listu

Základem strategie UCT je sestup stromem od kořene k listu na základě porovnávání win rate a počtů průchodů uzly – kandidáty. Algoritmus má snahu hlouběji prohledávat ty tahy, které se jeví jako perspektivnější. Na druhou stranu je potřeba prohledávat i ty tahy, které ještě nebyly prozkoumány vůbec, nebo byly prozkoumány málo, aby nedocházelo k přehlédnutí dobrých tahů, nebo chybnému vyhodnocení dobrého tahu jako špatného kvůli náhodnému neúspěchu na malém množství odsimulovaných her pro tento tah. Tento problém je označován jako udržování rovnováhy mezi využíváním a prozkoumáváním (anglicky "balance between exploitation and exploration").

Vyvážené procházení stromu je zajištěno porovnáváním hodnoty *UCTValue* všech synů zkoumaného uzlu a následným výběrem syna s nejvyšší hodnotou

UCTValue. Hodnota *UCTValue* pro zkoumaný uzel *u* je vypočtena následující formulí (Hollosi, Pahle, [2]):

$$UTCValue(u) = u.winrate + C \cdot \sqrt{\frac{log(u.otec.pruchody)}{u.pruchody}}$$

Hlavním měřítkem pro hodnotu *UCTValue* je win rate zkoumaného uzlu *u*, který zajišťuje využití slibných tahů, zatímco druhý term zvyšuje výslednou hodnotu u těch uzlů, které byly málo prozkoumány a pokud počet průchodů otcem uzlu dosáhne určité hranice, pak spolu se svým win rate může uzel v hodnotě *UCTValue* předstihnout i statisticky úspěšnější tah. Pokud se uzel (tedy tah, který uzel reprezentuje) ukáže v simulaci jako úspěšný, vzroste tím jeho win rate a bude s větší pravděpodobností brzy opět vybrán. Je-li naopak neúspěšný, klesne win rate i hodnota druhého termu a uzel bude muset pravděpodobně dlouho čekat, než bude opět vybrán.

Konstanta *C* upravuje hodnotu druhého termu na rozumný poměr k win rate, který je v této implementaci v rozmezí [0,1]. *C* tedy musí hodnotu druhého termu vhodně snížit, aby hlavním faktorem pro rozhodování byl win rate. S vyšší hodnotou konstanty *C* se rovnováha v procházení stromu přesouvá směrem k širšímu prohledávání, tedy vyšší pravděpodobnosti výběru málo prozkoumaných uzlů, naopak s nižší hodnotou se přesouvá směrem k hlubšímu prohledávání uzlů, které byly doposud úspěšné. Standardní hodnoty pro *C* jsou např. 0,2 nebo 0,1. Význam konstanty pro výsledné chování programu bude experimentálně otestován v dalších kapitolách. V přiloženém programu a pro potřeby testování se tato konstanta nazývá *UTCConst*.

Uvedená podoba vzorce pro výpočet *UCTValue* platí pro ty uzly *u*, ve kterých je na tahu bílý hráč. Černý počítá s opačným win rate, tedy první term má tvar *(1-u.winrate)*.

Vzhledem k vysokému větvícímu faktoru stromu hry je při omezeném času na přemýšlení možné přizpůsobit algoritmus očekávanému menšímu množství testů. V základní variantě UCT se používá pravidlo projít nejdřív každého syna jednou, což je přirozené v případě, že nepředpokládáme žádné znalosti specifické pro hru.

Chceme-li však prohledat některé uzly hlouběji, je vhodné povolit opakovaný průchod uzlem i v případě, že některý jeho bratr ještě nebyl vybrán ani jednou. Jinými slovy průchod stromem může skončit i v uzlu, který není listem. Jednoduchým způsobem, jak rozhodnout, zda pokračovat v sestupu stromem, nebo přejít k vytvoření nového uzlu, je přidělení 50% pravděpodobnosti každé z těchto dvou možností a náhodné rozhodnutí. S touto úpravou je stavěný strom užší, hrozí však přehlédnutí důležitých tahů. Proto je nutné zakázat toto vynechávání uzlů minimálně v kořeni stromu, případně na několika prvních úrovních, kde by se měly vzít v potaz všechny možné tahy.

3.3. Vytvoření a připojení nového uzlu

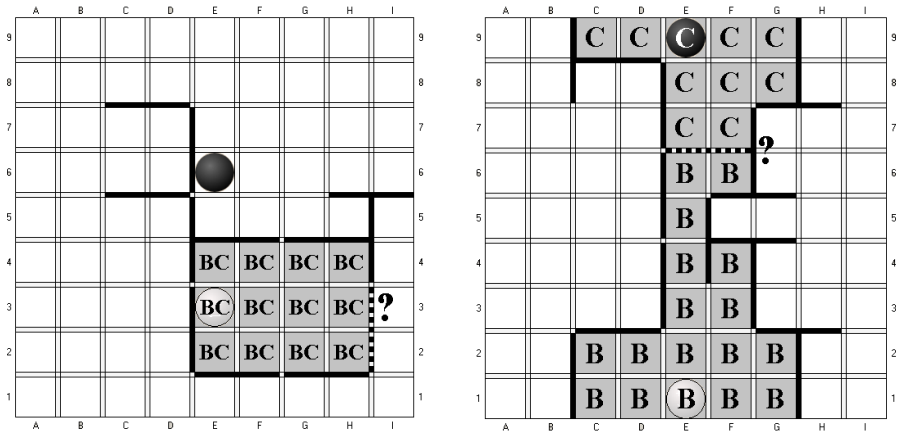
Jakmile je dokončen průchod stromu v paměti a je vybrán poslední uzel, je třeba přidat do stromu nový list. Ten je vybrán jako pseudonáhodný tah funkcí na výběr tahů v playoutu, na jejímž návrhu závisí jak realističnost playoutů, tak časová náročnost programu.

Výběr náhodného tahu je krok, který se vyskytuje jak při přidání nového uzlu, tak v simulaci hry mimo strom. Protože jde o mnohokrát opakovaný krok, je vhodné jej co možná nejvíce zrychlit. Jak již bylo zmíněno v předchozí kapitole, samotné ověření legality tahu vybraného z množiny všech možných tahů při prázdné hrací ploše bez překážek sjednocené navíc s tahy figurkou do čtyř možných směrů (nebo ekvivalentně vytvoření seznamu možných tahů v dané pozici) je netriviální úloha z důvodu nutnosti zachování průchodné cesty obou hráčů do cíle. Z tohoto důvodu je možné ušetřit čas na ověřování legality náhodného tahu tím, že seznam zakázaných tahů budeme udržovat v paměti nejen po dobu hledání náhodného tahu dané pozici, ale v rámci běhu programu až do konce hry.

Tento způsob trvalého vyloučení některých tahů můžeme jednoznačně použít pro tahy, které pokládají překážku na místo, kde by se překrývala s jinou překážkou. Vzhledem k tomu, že každá překážka je položena na plochu definitivně a nemůže až do konce hry změnit své místo, je totiž jisté, že tah, který je z důvodu překrývání překážek nemožný, bude nemožný i ve všech následujících pozicích až do konce hry.

Naopak ukládání do paměti nemožných tahů figurkou nemá smysl, neboť možnost táhnout figurkou určitým směrem se neustále mění v závislosti na poloze figurky a na položených překážkách. V implementaci programu použité pro tuto práci jsou tahy figurkou rozděleny pouze na 4 možnosti – podle směru pohybu. Alternativní provedení by mohlo být popisovat tah figurou jako pole, na které se má přesunout, ale i v tom případě by ukládání nemožných tahů nemělo smysl ze stejného důvodu.

Třetí skupinu nemožných tahů (nebo třetím pravidlem pro zákaz tahu) jsou tahy pokládající překážku, která by některému hráči zcela odřízla cestu do cíle. Protože překážky je možné v průběhu hry pouze přidávat, je jisté, že každá překážka, která by rozdělila (ve smyslu průchodnosti pro figury) desku na dvě nebo více částí, si tuto vlastnost ponechá až do konce hry. Může se však stát, že potenciálně odříznutá figura nebo figury opustí prostor, který by překážka po položení uzavřela a tím se její položení stane legální. Proto je možné uchovávat v paměti seznam těchto zakázaných tahů, ale aby měl použití v dalších pozicích, musí být ke každému tahu přiřazen seznam kritických polí zvlášť pro každého hráče, na kterých musí stát jejich figura, aby byl tah nemožný.

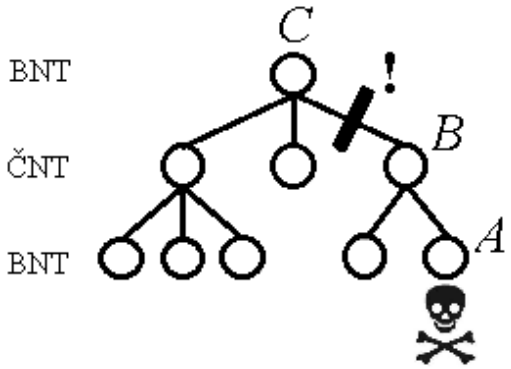


Obr 3.2: Zvýrazněná jsou kritická pole pro jednu překážku a pro každého hráče zvlášť

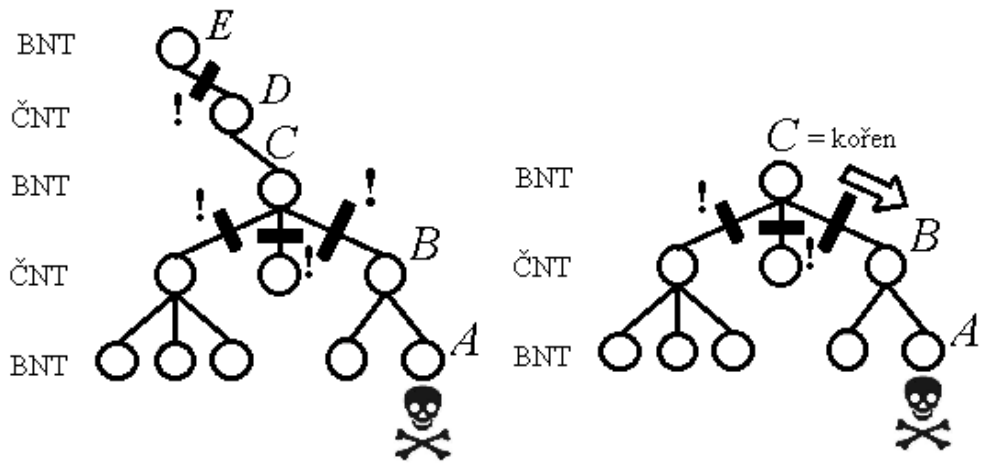
Seznam zakázaných tahů je udržován pro každý uzel zvlášť s tím, že každý uzel dědí tento seznam od svého otce a sám jej může rozšířit při neúspěšných pokusech o přidání nového syna (v programu jde o vygenerování náhodného čísla, které odpovídá tahu, který je proti pravidlům, ale zatím se to o něm neví)

Samozřejmostí je okamžité zavržení všech tahů překážkou v pozici, ve které aktivní hráč už nemá žádné překážky v zásobě. V takové pozici se rovnou berou v úvahu pouze tahy figurkou.

Další možnou optimalizací kompenzující slabou taktickou povahu MCTS a vysoký větvící faktor Quoridoru je detekce a zákaz prohrávajících tahů. Je-li při přidávání uzlu do stromu dosažena pozice *A*, která je prohrou aktivního hráče, je otec *B* (který je poslední a chybný tah aktivního hráče) uzlu *A* odstraněn ze stromu a navíc přidán do seznamu zakázaných tahů svého otce *C*, aby nemohl být znovu přidán a aby skrz něj neprobíhalo zbytečné prohledávání stromu. Průchod stromem pak pokračuje od otce *C* odebraného uzlu běžným způsobem. Výjimkou pro tuto akci je situace, kdy už jsou mezi zakázanými tahy pro uzel *C* všechny tahy. V takové situaci se za předpokladu, že *C* není kořen celá procedura rekurzivně opakuje s tím, že nyní je za prohranou pozici považován uzel *C*, je odříznut jeho otec *D* atd. Pokud je již *C* kořenem, znamená to, že aktivní hráč má na výběr pouze prohrávající tahy a nezbývá mu, než vybrat si jeden z nich.



Obr 3.3: Odříznutí uzlu vedoucího k prohře



Obr 3.4: Řešení v případě odříznutí všech synů

Pro úplnost: nemůže nastat situace, kdy by kořenem byl otec uzlu, který je tímto způsobem vyhodnocen jako prohraná pozice, a tudíž by postup nemohl pokračovat. Uzel vyhodnocovaný tímto způsobem jako prohraný totiž vždy reprezentuje pozici, ve které je na tahu aktivní hráč, v jeho otci (pokud existuje) je tedy na tahu neaktivní hráč, ale v kořeni je vždy na tahu aktivní hráč.

Nově přidaný list získá první data pro statistiky z playoutu, který jím projde při jeho přidání.

3.4. Dohrání partie náhodnými tahy

Z nově přidaného uzlu se spustí náhodný průchod hrou až do konce partie. Toto se již neodehrává ve stromu hry v paměti, ale pod ním. Náhodnou simulaci (playout) provede funkce, která na vstupu dostane výchozí pozici (reprezentovanou přidaným uzlem) a vrátí pouze konečný výsledek partie pro započítání do statistik. Konkrétní tahy v této fázi nejsou důležité, jsou však vybírány pseudonáhodně heuristickou metodou použitou i při vytváření nového uzlu, popsanou v 3.3. Uchovávání seznamu zakázaných tahů je v této fázi zvlášť efektivní, naopak detekci prohrávajících tahů použít nelze, protože neexistuje žádný strom, na kterém by se prořezávání provádělo.

Pro efektivní fungování programu je potřeba provést co největší množství playoutů. Zároveň je náhodná simulace nejnáročnějším cyklicky opakovaným prvkem algoritmu. Proto je důležité, aby její provedení bylo pokud možno rychlé a nenáročné.

Počet tahů v jednom playoutu může být teoreticky nekonečně velký, figurky se totiž mohou pohybovat tam a zpět po hrací desce aniž by se jedna z nich dostala do cíle po mnoho tahů a pokud jsou tahy vybírány zcela náhodně, je velmi pravděpodobné, že tomu tak bude.

Prvním možným řešením tohoto problému, které se nabízí, je použití heuristické ohodnocovací funkce, která po odehrání určitého množství náhodných tahů provede rychlý odhad výsledku na základě aktuální pozice. Ohodnocovací funkce sama o sobě je komplexní problém a proto bude podrobena rozboru ve vlastní kapitole.

Použití ohodnocovací funkce po určitém množství náhodně odehraných tahů řeší problém s délkou playoutů. Další otázkou je kvalita playoutů. Od playoutů se očekává, že poměr počtů výher obou hráčů bude odpovídat situaci na hrací desce. Tento odhad však ztrácí přesnost, pokud jeden hráč postupuje přímo k cíli a druhý se od něj naopak vzdaluje, nebo pochoduje na místě. Řešením tohoto problému je nahrazení náhodného tahu pseudonáhodným tahem vybraným na základě pomocných kritérií, stejných nebo podobných jako při použití ohodnocovací funkce. Tato heuristika výrazně zvýší náročnost jednoho playoutu, což je v rozporu se základní myšlenkou co největšího množství playoutů. Na druhou stranu ale slibuje víc realistické playouty a zvýšení přesnosti výsledků playoutů. Proto je její použití otázkou, která bude zodpovězena experimentálně.

Pro potřeby testování budeme tuto heuristiku nazývat "chytré playouty" a její konkrétní návrh je následující: Při provádění náhodné simulace se každý sudý tah každého hráče provede krok figurkou směrem k cíli nejkratší cestou (tah se najde algoritmem vlny, viz 4. kapitolu). Liché tahy v pořadí (v rámci tahů jendoho hráče) se vybírají zcela náhodně. Tímto postupem je zaručen postup obou hráčů směrem k cíli, zároveň je šance projít všechny možné pozice vzniklé pokládáním překážek a časová náročnost simulace se nezvýší příliš drasticky.

3.5. Zpětná propagace

Výsledek simulace z 3.4 se započítá zpětně do statistik uzlu, ze kterého byla simulace spuštěna a dále rekurzivně do jeho otce atd. až do kořene stromu. Úprava se tedy týká všech uzlů (tahů), které jsou ve stromu a které byly součástí této partie, která skončila posledním playoutem.

Každému z těchto uzlů se:

1. Zvýší počet průchodů o 1
2. Zvýší skóre o výslednou hodnotu playoutu (více níže)
3. Aktualizuje win rate. Za win rate se dosadí podíl skóre a počtu průchodů

Skóre uzlu je zobecněnou variantou počtu výher. Zatímco při rozlišování výsledku playoutu pouze na výhru/prohru se může přičíst pouze 0 nebo 1, V módu zkrácených playoutů při použití ohodnocovací funkce je možné vyjádřit pravděpodobný výsledek partie reálným číslem z intervalu [0,1], kde hodnoty blízké nule znamenají téměř jisté vítězství černého, *0,5* znamená vyrovnanou hru a hodnoty blížící se *1* značí vítězství bílého. Zjednodušeně řečeno skóre se snaží vyjádřit pravděpodobnost vítězství bílého hráče. Skóre je vypočítáno v rámci ohodnocovací funkce, viz kapitola 5.

V případě, že se playouty dohrávají až do konce (bez odhadu po určitém počtu náhodných tahů) se skutečně připočítává ke skóre vždy 0 nebo 1. Obě možnosti jsou implementovány v přiloženém programu a jejich použití závisí na konfiguraci.

4. Ohodnocovací funkce

Ohodnocovací funkce (anglicky evaluation function) je důležitou součástí většiny algoritmů na procházení stromů her. Je to funkce, která každé pozici přiřazuje jednu reálnou hodnotu, která vyjadřuje, jak dobře si hráči v této pozici stojí. Ačkoliv MCTS v základní variantě nevyžaduje použití ohodnocovací funkce, je možné ji použít pro zkrácení playoutů – odhad výsledku v určité fázi simulované hry, díky kterému se nemusí náhodná hra dohrávat až do konce a ušetří se tak čas, který by trvalo dohrání.

Ohodnocovací funkce by neměla být časově příliš náročná, aby sama nespotřebovala čas který se ušetří ukončením prohledávání v dané pozici. Zároveň by měla v nekoncových pozicích co nejvěrněji odrážet reálnou pravděpodobnost výsledku partie. Funkce se proto většinou optimalizuje jako kompromis mezi přesností a časovou náročností. (Glendenning, [2])

Je zřejmé, že ohodnocovací funkce nemůže nikdy fungovat zcela přesně a ani to není jejím účelem. Pokud by totiž dokázala u všech pozic neomylně určit, které jsou lepši než jiné, znamenalo by to vyřešení celé hry. Ohodnocovací funkce jsou zpravidla heuristické a počítají s různými vlastnostmi dané pozice, které dohromady skládají výslednou hodnotu. Nejjednodušší ale používaná metoda je skládání vlastností lineární kombinací. Každá měřená vlastnost f je násobena váhou w, aby tak vynikly ty vlastnosti, které se zdají být "důležitější".

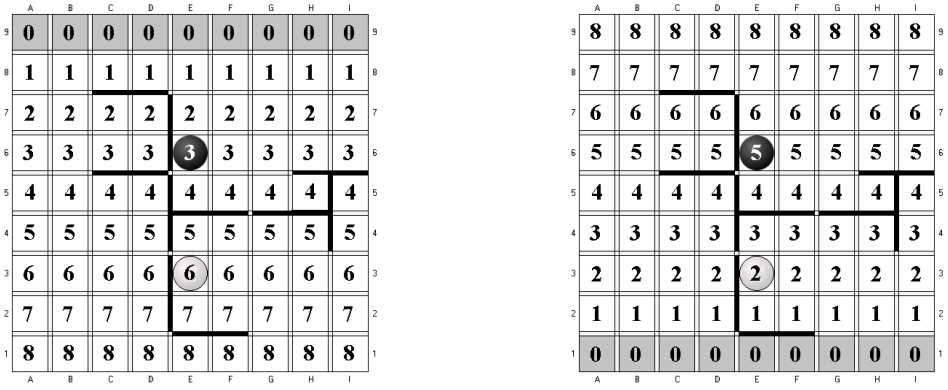
Ohodnocovací funkci E pro pozici s, vlastnosti f a váhy w tedy můžeme definovat vzorcem:

$$E(s) = \sum_i w_i \cdot f_i$$

Použití jednotlivých vlastností a jejich vzájemné vyvážení je přitom otevřenou otázkou.

Primitivním způsobem, jak ohodnotit postavení hráčů v pozici, je spočítat počet řad, které ještě musí každý hráč překonat, aby se dostal do cíle, a pro ohodnocení pozice použít rozdíl těchto dvou hodnot. Tato metoda je na první pohled

krátkozraká, protože nebere ohled na položené překážky. Na druhou stranu je tento výpočet velmi jednoduchý, známe-li pozice obou hráčů, rychle (v konstantním čase) z nich spočítáme rozdíl vzdáleností.

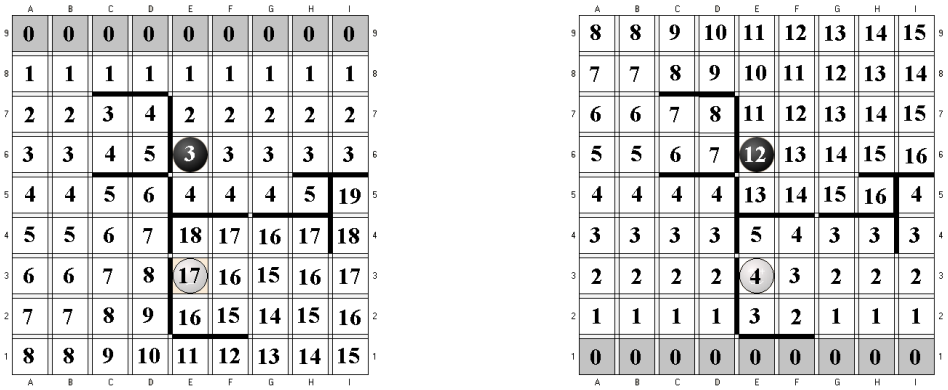


Obr. 4.1: Měření vzdálenosti na počet řad zbývajících do cíle

Pro zjištění rozdílu počtů řad zbývajících do cíle stačí vzít součet ypsilonových pozic hráčů, který je tomuto rozdílu roven. Přesněji

$$f = bily.y + cerny.y - 8$$

Realističtějším způsobem ohodnocení pozice je počítání počtu tahů, který oběma hráčům zbývá do cíle. Hráč, který je blíže k cíli je na tom zřejmě lépe. Jako měřitelná vlastnost poslouží rozdíl těchto hodnot.



Obr. 4.2: Měření vzdálenosti na počet tahů zbývajících do cíle

Na *Obr. 4.2* je znázorněno ohodnocení všech polí hodnotou odpovídající vzdálenosti od cíle, vypočtené algoritmem vlny, zvlášť pro bílého a zvlášť pro černého hráče. Bílý stojí ve vzdálenosti 17 od svého cíle, černý 12 od svého. Tato

vlastnost *f* pozice na obrázku má tedy hodnotu -5. (Bílý hráč zaostává 5 polí za černým.) Vzdálenost se počítá prohledáváním do šířky ("vlnou"). Začátek algoritmu vlny v cíli umožňuje oproti vlně vycházející z pole obsazeného figurkou kromě ohodnocení pozice i rychle rozhodnout, které ze sousedních polí je nejblíže k cíli, což lze použít pro heuristickou úpravu strategie playoutu.

Dalším nezanedbatelným faktorem, který je třeba vzít v úvahu při ohodnocování pozice, je počet zbývajících překážek obou hráčů. Překážky jsou stejně cenné, proto stačí rozdíl těchto počtů. Je zřejmé, že hráč kterému zbývá více překážek, je ve výhodě. Jedinou otázkou ohledně naší implementace této vlastnosti pozice tak zůstává váha překážek vzhledem k váze vzdáleností.

V přiloženém programu je ohodnocovací funkce implementována jako výpočet rozdílu vzdáleností podle počtu tahů zbývajících do cíle (*Obr. 4.2*) sečtenou s počtem zbývajících překážek obou hráčů váženým konstantou *zarazkaConst*

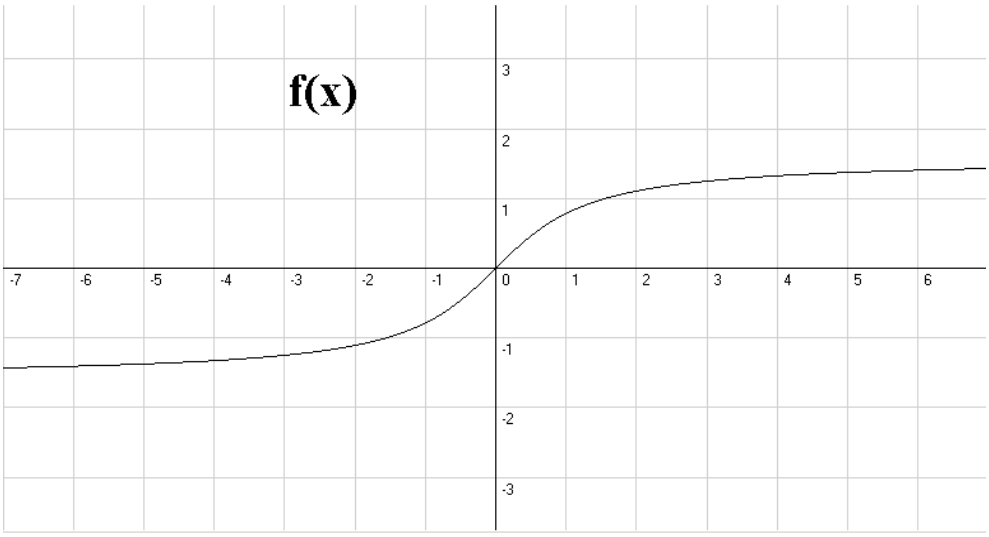
Vzorec pro funkci tedy vypadá takto:

$$E(s) = d(s, cerny) - d(s, bily) + (zarazkaConst \cdot (bily.prekazky - cerny.prekazky))$$

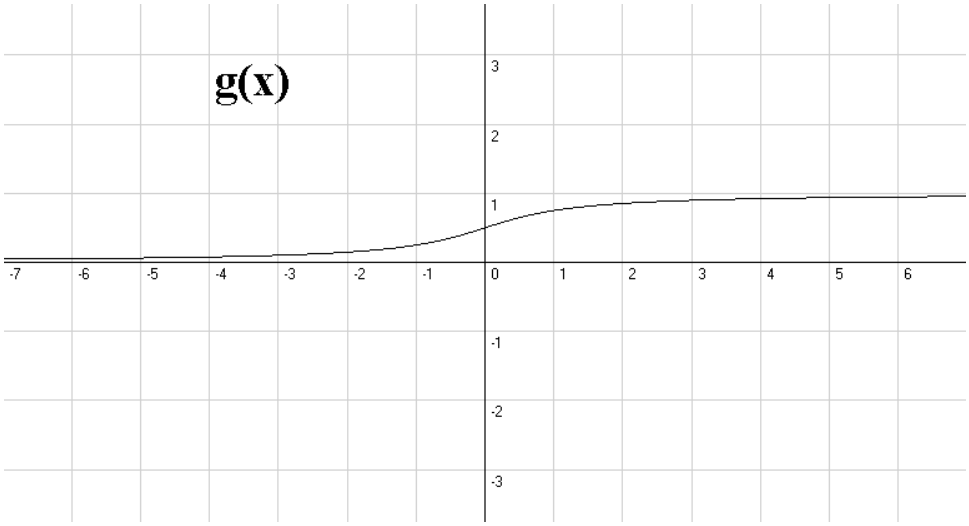
Kde *d(s, hrac)* je vzdálenost hráče *hrac* od cíle v pozici *s*.

Uzly stromu však pro výpočet win rate nepotřebují znát rozdíl vzdáleností na konci partie, ale hodnotu skóre v intervalu [0,1], která je slučitelná s rozlišováním pouhých výher a proher u playoutů s hodnocením 0 za prohru (bílého) a 1 za výhru (bílého), viz 3.5. Při tom je logické, aby samotná výhra v partii byla ohodnocena nejkrajnější možnou hodnotou ohodnocovací funkce.

Převod rozdílu vzdáleností na skóre provádí speciální funkce, která je prostředníkem mezi ohodnocovací funkcí a UCT. Implementována je jako součást ohodnocovací funkce. Tato funkce dostává na vstup hodnotu rozdílu většinou v intervalu cca [-10; 10], teoreticky je však možné dojít do pozice s mnohem větším rozdílem. Proto je vhodné, aby její definiční obor byl $\mathbb{R}$. Dále potřebujeme, aby tato funkce byla rostoucí, aby její hodnota v *0* byla *0,5* a aby její limity v $\pm\infty$ byly *0* a *1*. Těchto vlastností můžeme dosáhnout vhodným použitím funkce arkus tangens.



Obr. 4.3: $f(x) = tan^{-1}\, x$



Obr. 4.4: $f(x) = \frac{tan^{-1}\, x}{\pi} + 0{,}5$

Vydělení a přičtení konstant zajistí správné limity funkce, jak ukazuje funkce *g(x)* na *Obr. 4.3*. Otázkou však zůstává správná rychlost růstu funkce. Ta je nastavitelná parametrem *skoreConst*, kterým se násobí x uvnitř funkce arkus tangens. Konečný vzorec pro funkci na výpočet skóre je tedy:

$$skore(x) = \frac{tan^{-1}(skoreConst \cdot x)}{\pi} + 0{,}5$$

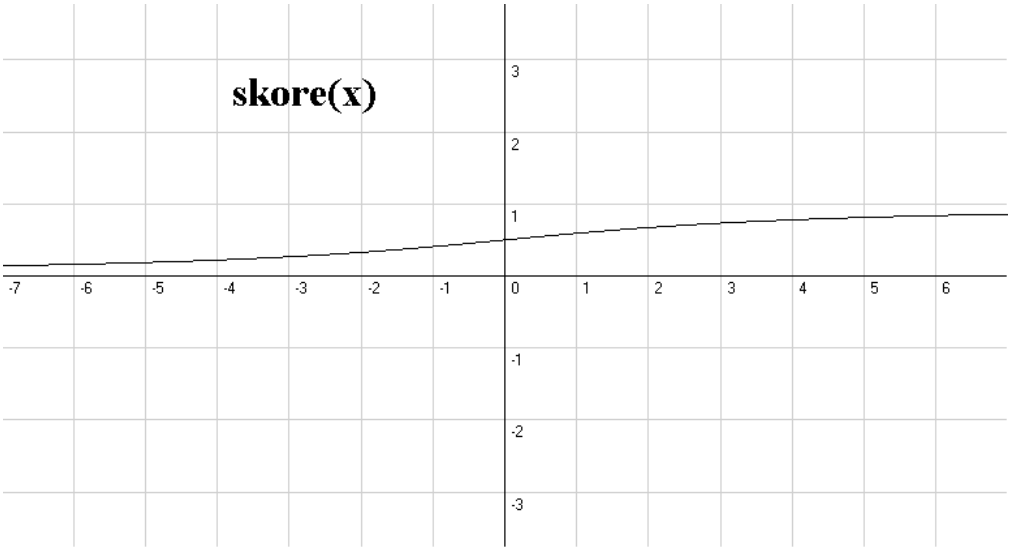
Krátký výčet hodnot funkce pro různé konstanty *skoreConst*:

$\frac{x}{sCon}$	0	1	2	3	4	5	6	7	8	9	10
0,1	0,5	0,53	0,56	0,59	0,62	0,65	0,67	0,69	0,71	0,73	0,75
0,25	0,5	0,58	0,65	0,7	0,75	0,79	0,81	0,83	0,86	0,87	0,88
0,5	0,5	0,65	0,75	0,81	0,85	0,88	0,9	0,91	0,92	0,93	0,94
1	0,5	0,75	0,85	0,89	0,92	0,94	0,95	0,95	0,96	0,96	0,97

Hodnoty pro záporné rozdíly *x* nejsou uvedeny, jsou však symetrické s kladnými se středem symetrie [0; 0,5], a proto platí:

$$skore(x) - 0{,}5 = 0{,}5 - skore(-x)$$

$$skore(-x) = 1 - skore(x)$$



Obr. 4.5: graf funkce skore(x) při skoreConst = 0,3

Jinou možností, jak převádět rozdíl vzdáleností na skóre, by bylo použití sigmoidové funkce dané předpisem $P(t) = \frac{1}{1+e^{-t}}$, která stejně jako arkus tangens má požadované vlastnosti.

5. Analýza algoritmu

5.1. Vliv heuristik a optimalizací na efektivitu a náročnost

V předchozích kapitolách byla navržena aplikace MCTS na Quoridor a k základní variantě MCTS bylo navrženo několik optimalizací a heuristik. Tato kapitola se bude zabývat jejich přínosem pro algoritmus. Účelem navržených optimalizací je zpravidla snížení časové náročnosti využitím redundancí. Stejné výpočty, které se provádějí opakovaně je možné ukládat v paměti a některé výpočty, které za určitých okolností nejsou využity je možné uvolnit. Při dosažení nižší časové náročnosti jednoho cyklu UCT je možné za čas, který má algoritmus na svůj běh k dispozici, provést více playoutů, což vede k lepšímu výběru dalšího tahu. Naproti tomu heuristiky nemají na náročnost algoritmu vliv, ale snaží se zvyšovat přímo jeho efektivitu, a to na základě určitých předpokladů, jak by měl algoritmus správně postupovat. Tyto předpoklady však nejsou na rozdíl od optimalizací podloženy matematickými tvrzeními, a proto je třeba jejich pozitivní dopad ověřit experimentálně.

Každá heuristika i optimalizace s sebou navíc nese určitou režii, která může být v nejhorším případě vyšší, než její přínos a "vylepšený" algoritmus pak dosahuje horších výsledků, než bez této úpravy. Účinnost všech heuristik a optimalizací proto bude experimentálně ověřena.

Než přejdeme k samotnému testování, shrneme si krátce seznam vlastností a parametrů algoritmu, které chceme otestovat a co od nich očekáváme:

č.	Vlastnost	Typ	Očekávaný efekt
1	Počet playoutů na jeden tah	vnější omezení	Síla algoritmu roste s počtem playoutů, otázkou je, jak rychle.
2	Konstanta *UTCConst* (kapitola 3.2)	heuristika	Zvýšení efektivity algoritmu optimalizací konstanty (lépe vyvážený strom)

3	Maximální délka playoutu (kapitola 3.4)	heuristika	Více provedených playoutů za stejný čas
4	Konstanta *zarazkaConst* (kapitola 4)	heuristika	Zvýšení efektivity algoritmu optimalizací konstanty (přesnější evaluace)
5	Konstanta *skoreConst* (kapitola 4)	heuristika	Zvýšení efektivity algoritmu optimalizací konstanty (evaluace)
6	Vynechávání uzlů od určité hloubky (kapitola 3.2)	heuristika	Hlubší průzkum nadějných uzlů
7	Chytré simulace (kapitola 3.4)	heuristika	Realističtější playouty
8	Pamatování překrývajících se překážek (kapitola 3.3)	optimalizace	Více provedených playoutů za stejný čas
9	Pamatování blokujících překážek (kapitola 3.3)	optimalizace	Více provedených playoutů za stejný čas
10	Detekce prohrávajících tahů (kapitola 3.3)	optimalizace	Vyšší efektivita + více provedených playoutů za stejný čas

5.2. Pevný bod a podmínky pro testování

Pro každou ze zkoumaných vlastností (parametrů) bude provedena řada testovacích her, které ověří splnění nebo nesplnění očekávaného efektu. V případě parametrů, které mohou nabývat jen dvou hodnot (aktivní / neaktivní) bude

provedena jen jedna série testovacích her, u parametrů pro které chceme najít optimální hodnotu, bude sérií několik.

Testování provede program MCTS-Quoridor v testovacím módu, který je jeho součástí. V testovacím módu program umožňuje opakované hraní partií algoritmu sám proti sobě, přičemž pro každého hráče (bílý, černý) mohou být nastaveny jiné parametry algoritmu. Tyto parametry se pro oba hráče nastavují v konfiguračním souboru. Po spuštění testování začne program hrát a zobrazovat výsledky odehraných partií až do dosažení požadovaného počtu testovacích partií.

Při testování každého jednotlivého parametru bude zvolen tzv. pevný bod – kompletní sestava parametrů pro umělou inteligenci, proti kterému se bude testovat zkoumaný parametr. Jako pevný bod zpravidla stanovíme černého hráče a budeme zkoumat úspěšnost bílého proti němu. Pevný bod tím spolu s úspěšností umělé inteligence nastavené zkoumaným parametrem vytvoří měřítko úspěšnosti tohoto parametru.

V každé sérii bude odehráno 100 testovacích partií a na základě jejich výsledku bude zhodnocen efekt nastavení konkrétního parametru. Aby bylo testování korektní, je třeba zajistit rovné podmínky pro obě strany, ve smyslu všech podmínek a okolností kromě parametrů umělé inteligence. Proto bude v každé sérií v 50 partiích začínat bílý a v 50 černý (obecně při jakémkoliv počtu testovacích partií začíná každou sudou v pořadí černý), aby nebyl bílý hráč z(ne)výhodněn právem a povinností prvního tahu. Obecně zkoumáme parametry bez ohledu na to, zda jsou pro začínajícího, nebo druhého hráče.

Aby byly porovnatelné výsledky testů, při kterých je běh algoritmu omezen časovým limitem, je pro tyto testy nutné použit vždy stejný hardware, jinak by se v různých sériích mohl provádět jiný počet playoutů na tah při stejném nastavení algoritmu a výsledky by tak byly irelevantní. Toto se netýká těch testování při kterých je algoritmus omezen jen na počet playoutů a ne na čas. V takovém případě zřejmě nehraje hardware roli. V případě testování omezeného na čas je pro potřeby této práce vždy použit procesor Intel Pentium T4300 s taktovací frekvencí 2,1 GHz.

5.3. Výsledky experimentů a význam jednotlivých parametrů

Experiment č. 1: Počet playoutů na jeden tah

V tomto experimentu zjišťujeme, jak moc algoritmus citlivý na změnu počtu playoutů, které provádí v jednom běhu. Za pevný bod použijeme nastavení 2500 playoutů na každý tah pro černého hráče a budeme úspěšnost bílého hráče při různém nastavení počtů playoutů pro sebe. Ostatní parametry jsou pro oba hráče stejné a konkrétně mají tyto hodnoty:

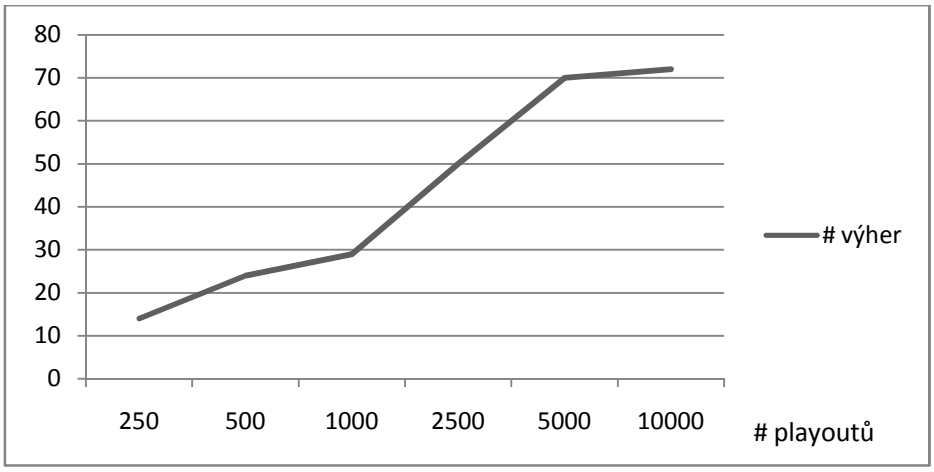
Parametr č.	1	2	3	4	5	6	7	8	9	10
Hodnota	2500	0,1	0	1	0,3	OFF	OFF	OFF	OFF	OFF

Nezkoumané parametry budou pro oba hráče vždy stejné i ve všech dalších experimentech, proto k informaci o nastavení UI postačí tato tabulka s popisem pevného bodu.

Pokud je v tabulce hodnot pro testovaný parametr použita i hodnota parametru pevného bodu, předpokládá se v této sérii 50% úspěšnost bez testování. UI totiž hraje sama proti sobě při stejném nastavení a proto relativní úspěšnost obou hráčů konverguje k 50% při počtu testovacích her jdoucím do nekonečna.

Výsledky experimentu č. 1:

# playoutů	250	500	1000	2500	5000	10000
# výher	14	24	29	50	70	72



Obr 5.1: graf úspěšnosti UI v závislosti na počtu playoutů

Výsledky experimentu potvrzují očekávání, že s počtem playoutů na tah roste síla algoritmu. Je také zajímavé, že poměr vítězství obou hráčů přibližně ale přece jen odpovídá poměru provedených playoutů.

Experiment č. 2: konstanta *UCTConst*

V tomto experimentu hledáme optimální hodnotu konstanty *UCTConst*, která má za úkol udržovat rovnováhu mezi využíváním a prohledáváním. Čím vyšší *UCTConst*, tím je stavěný strom širší. Naopak čím nižší, tím je strom užší a hlubší.

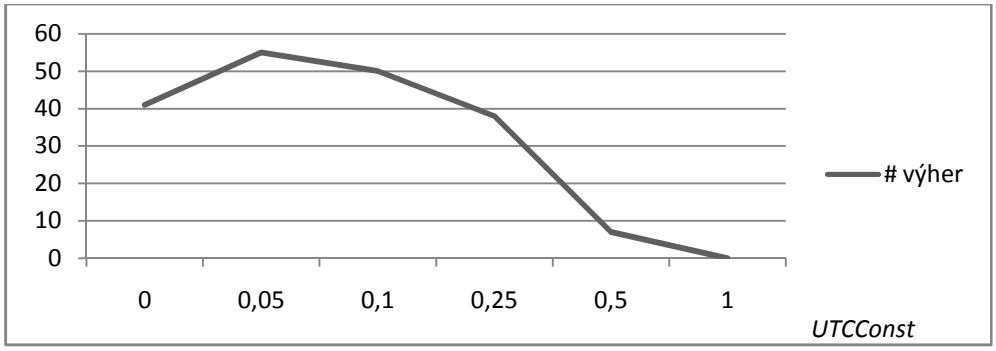
Nastavení pevného bodu:

Parametr č.	1	2	3	4	5	6	7	8	9	10
Hodnota	1,5s	0,1	2	1	0,3	OFF	OFF	OFF	OFF	OFF

Jako omezení algoritmu (parametr č. 1) v tomto případě není určen počet playoutů ale čas, v tomto případě 1,5 sekundy na tah. Při popsaných podmínkách se v tomto čase stihne provést cca 2500-3000 playoutů.

Výsledky experimentu č. 2:

UCTConst	0	0,05	0,1	0,25	0,5	1
# výher	41	55	50	38	7	0



Obr 5.2: graf úspěšnosti UI v závislosti na konstantě UTCConst

Výsledek experimentu ukazuje, jako optimální hodnotu *UCTConst* 0,05. (pro tohoto soupeře a tento čas)

Experiment č. 3: maximální délka playoutu

V tomto experimentu prověříme, zda je výhodnější provádět playouty až do konce, nebo je po určitém počtu tahů ukončit a udělat odhad. Kratších playoutů bude možné provést více za stejný čas, ale budou méně realistické. V tomto experimentu budeme poprvé měřit zároveň úspěšnost her proti pevnému bodu i průměrný počet provedených playoutů, který se podaří v jednom běhu algoritmu provést.

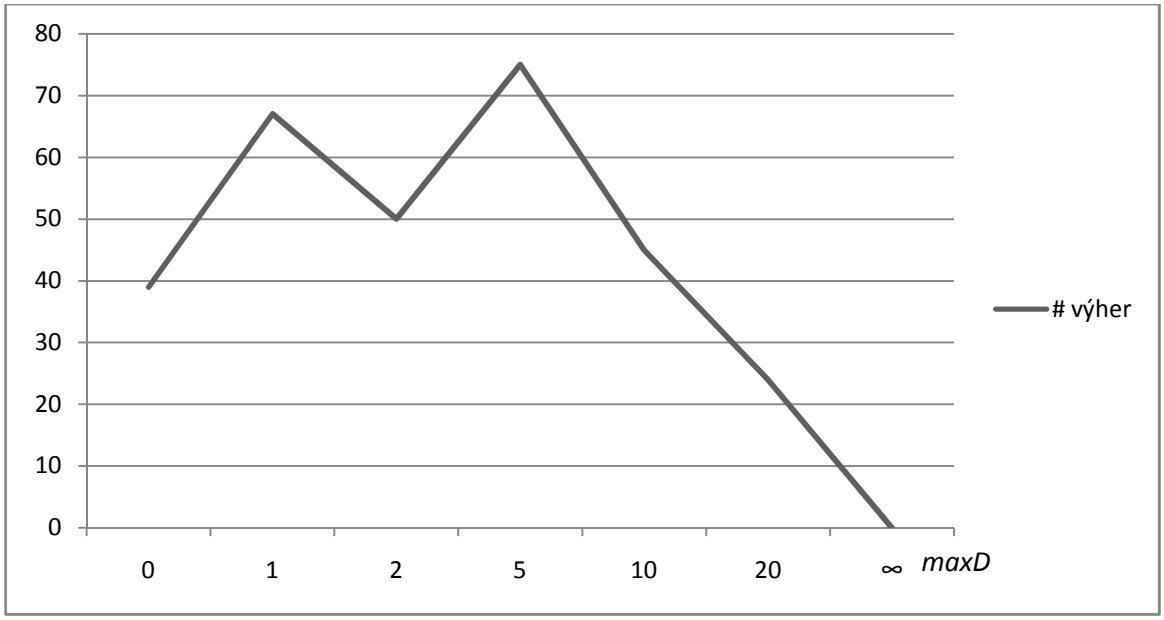
Pevný bod:

Parametr č.	1	2	3	4	5	6	7	8	9	10
Hodnota	2s	0,1	2	1	0,3	OFF	OFF	OFF	OFF	OFF

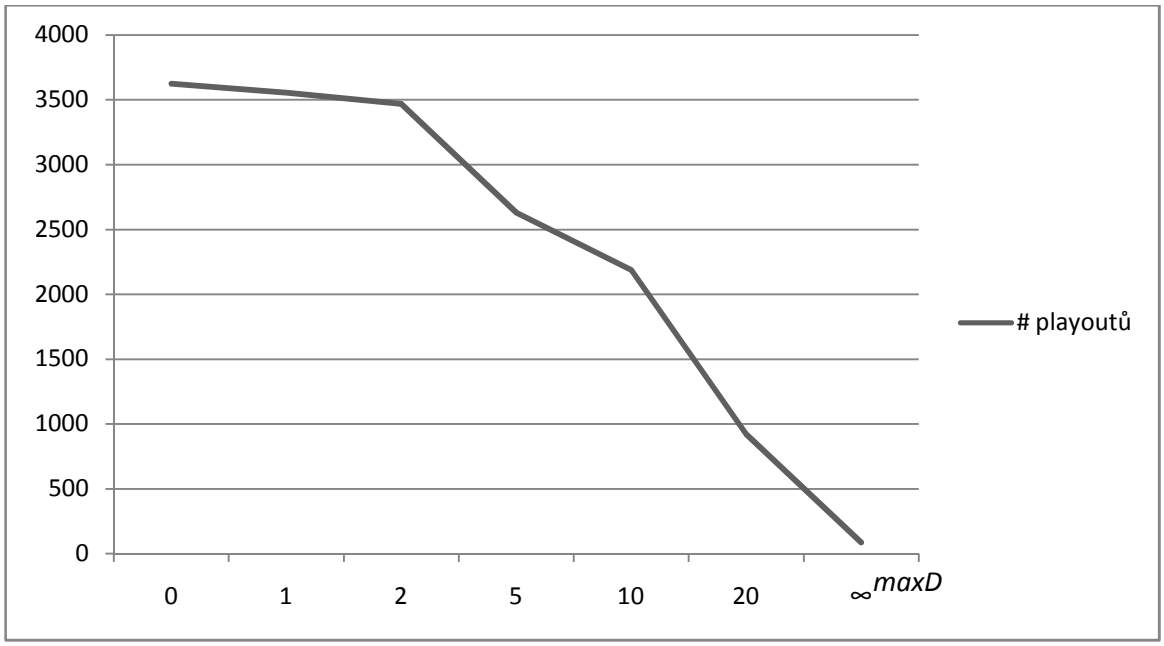
Parametr pro maximální délku simulace označujeme v tabulce *maxD*.

Výsledky experimentu č. 3:

maxD	0	1	2	5	10	20	∞
# výher	39	67	50	75	45	24	0
# playoutů	3624	3555	3469	2630	2186	920	86



Obr 5.3: graf úspěšnosti UI v závislosti na délce playoutů



Obr 5.4: graf závislosi počtu provedených playoutů na jejich délce

Klesající počet provedených playoutů s jejich rostoucí délkou je očekávaný výsledek. Zajímavý je však "zub", který se vytvořil na grafu úspěšnosti algoritmu. Vysoká úspěšnost při hodnotě parametru 1 je pravděpodobně způsobena statistickou odchylkou.

Experiment č 4: konstanta *zarazkaConst*

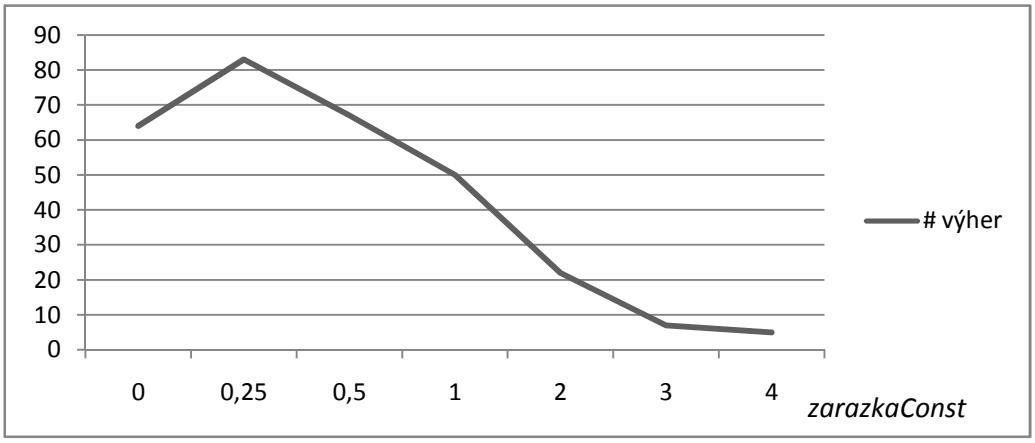
V tomto experimentu hledáme optimální hodnotu konstanty *zarazkaConst*, která vyvažuje hodnotu zbývajících překážek hráčů s tahy, které jim zbývají do cíle. Při hodnotě konstanty 1 má zarážka pro ohodnocovací funkci stejnou váhu jako 1 tah zbývající do cíle.

Pevný bod:

Parametr č.	1	2	3	4	5	6	7	8	9	10
Hodnota	2s	0,1	5	0,25	0,3	OFF	OFF	OFF	OFF	OFF

Výsledky experimentu č. 4:

zarazkaConst	0	0,25	0,5	1	2	3	4
# výher	64	83	67	50	22	7	5



Obr 5.5: graf úspěšnosti UI v závislosti na konstantě zarazkaConst

Z testování vychází přesvědčivě nejlépe hodnota 0,25 jako optimální pro konstantu *zarazkaConst*. Je zajímavé, že se vyplácí hodnotit překážku tak nízce, když v ideálním případě můžeme jejím položením prodloužit soupeřovi vzálenost o 4 pole (teoreticky i více). Měření však ukazuje, že se nevyplácí s takovými případy počítat plošně.

Experiment č. 5: konstanta *skoreConst*

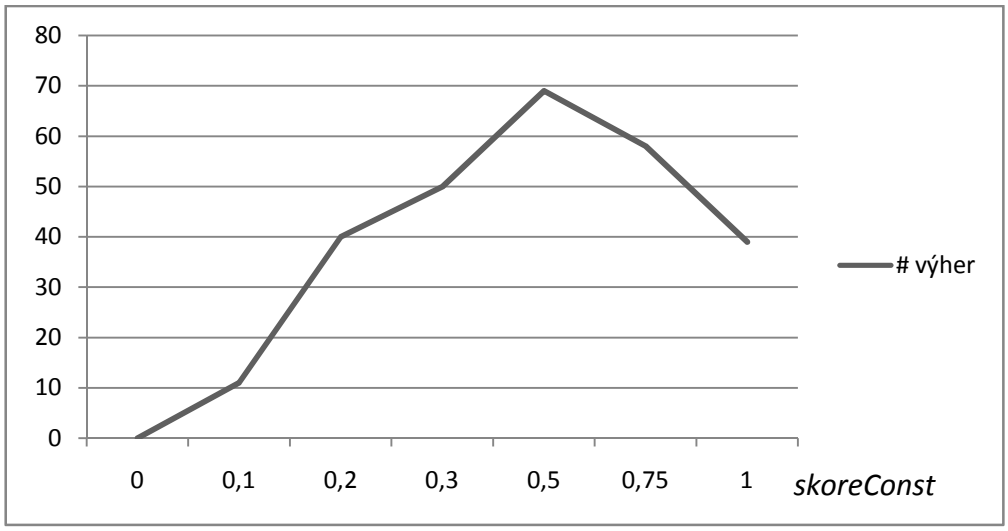
V tomto experimentu hledáme optimální hodnotu konstanty *skoreConst*, která ovlivňuje převod rozdílu vzdáleností na skóre, které se propaguje do stromu. Nižší hodnota *skoreConst* znamená pomalejší růst skóre s rostoucím rozdílem vzdáleností, naopak vyšší *scoreConst* znamená citlivější změny skóre už při malých rozdílech vzdáleností.

Pevný bod:

Parametr č.	1	2	3	4	5	6	7	8	9	10
Hodnota	2s	0,1	5	0,25	0,3	OFF	OFF	OFF	OFF	OFF

Výsledky experimentu č. 5:

skoreConst	0	0,1	0,2	0,3	0,5	0,75	1
# výher	0	11	40	50	69	58	39



Obr 5.6: graf úspěšnosti UI v závislosti na konstantě skoreConst

Z testování vychází přesvědčivě nejlépe hodnota 0,5 jako optimální pro konstantu *skoreConst*.

Experiment č. 6: Vynechávání uzlů od určité hloubky

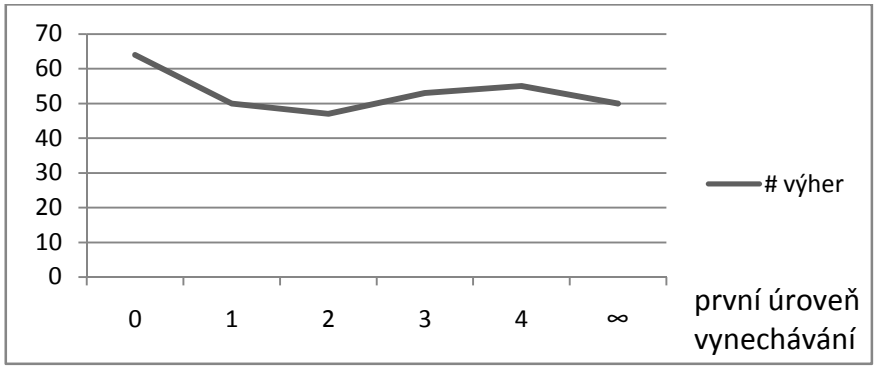
Vtomto expetimentu prověříme, zda se vyplatí procházet ve stromu uzly více než jednou i v případě, že existuje bratr tohoto uzlu, který ještě nebyl navštíven ani jednou. Tento bratr je tím vynechán a vlastnost algoritmu se proto nazývá vynechávání uzlů. Vynechávání je navíc možné povolit až v určité hloubce, například proto aby bylo jisté že každý potomek kořene bude navštíven alespoň jednou.

Pevný bod:

Parametr č.	1	2	3	4	5	6	7	8	9	10
Hodnota	2s	0,1	5	0,25	0,5	OFF	OFF	OFF	OFF	OFF

Výsledky experimentu č. 6:

Minimální hloubka pro vynechávání	0	1	2	3	4	∞
# výher	64	50	47	53	55	50



Obr 5.7: graf úspěšnosti UI v závislosti na hloubce vynechávání

Výsledné úspěšnosti algoritmu kolem 50% vnucují myšlenku, že na hloubce povolení vynechávání prakticky nezáleží. Při povolení vynechávání už v kořeni je však přece jen znát zlepšení, které pravděpodobně není jen statistickou odchylkou.

Experiment č. 7: Chytré simulace

V tomto experimentu prověříme účinnost mechanismu chytrých simulací popsaných v kapitole 3.4. Místo zcela náhodných tahů budou hráči v simulacích dělat takové tahy, které se na první pohled jeví lepší. Tím by se měla zvýšit realističnost playoutu, na druhou stranu hledání lepších tahů zabere čas a méně času znamená méně playoutů. Podívejme se tedy, zda se tato heuristika vyplati.

Pevný bod:

Parametr č.	1	2	3	4	5	6	7	8	9	10
Hodnota	2s	0,1	5	0,25	0,5	0	OFF	OFF	OFF	OFF

Výsledky experimentu č. 7:

Chytré simulace	OFF	ON
# výher	50	27
# playoutů	2652	1582

Chytré simulace dopadly v experimentu nečekaně špatně. Lepší odhad pozice evidentně nedokáže vynahradit velkou časovou ztrátu, kterou heuristika způsobila. Z poměru průměrných počtů playoutů na tah je vidět, že sama heuristika spotřebovala polovinu času spotřebovaného celým zbytkem algoritmu. Pro další testování proto zůstane nastavení této vlastnosti na OFF.

Experiment č. 8: Pamatování překrývajících se překážek

V tomto experimentu prověříme účinnost optimalizace, pamatování zakázaných tahů, která má přinést zrychlení při vyhledávání náhodných tahů, ale zase bude spotřebovávat čas při čtení seznamu těchto tahů a navíc zvýší paměťovou náročnost udržování stromu. Vyplatí se?

Pevný bod:

Parametr č.	1	2	3	4	5	6	7	8	9	10
Hodnota	2s	0,1	5	0,25	0,5	0	OFF	OFF	OFF	OFF

Výsledky experimentu č. 8:

Pamatování překrývajících se překážek	OFF	ON
# výher	50	62
# playoutů	2800	4431

Zlepšení algoritmu touto optimalizací je jasně zřetelné, spíše než samotnou úspěšností počtem provedených playoutů na tah, který se zvýšil více než o polovinu.

Experiment č. 9: Pamatování blokujících překážek

Povaha tohoto experimentu je velmi podobná předchozímu, ovšem z technického hlediska je uchovávání porovnávání blokujících překážek značně složitější než překrývajících a navíc se ve hře vyskytují méně často. Z tohoto důvodu se dá očekávat, že zlepšení nebude tak výrazné (pokud vůbec nějaké).

Pevný bod:

Parametr č.	1	2	3	4	5	6	7	8	9	10
Hodnota	2s	0,1	5	0,25	0,5	0	OFF	ON	OFF	OFF

Výsledky experimentu č. 9:

Pamatování blokujících překážek	OFF	ON
# výher	50	44
# playoutů	4368	4443

Očekávané zlepšení v počtu playoutů opravdu nastalo a podle očekávání je také velmi malé. Algoritmus s aktivovanou optimalizací sice dosáhl menšího počtu výher než soupeř, ale vzhledem k tomu, že všechny parametry kromě počtu playoutů byly pro oba hráče stejné, můžeme říct, je to důsledek náhody.

Experiment č. 10: Detekce prohrávajících tahů

Posledním experimentem bude prověření účinnosti detekce a odříznutí prohrávajících tahů ze stromu. Ve smyslu testování a očekávaných výsledků jde o podobnou optimalizaci, jako byly předchozí dvě.

Pevný bod:

Parametr č.	1	2	3	4	5	6	7	8	9	10
Hodnota	1s	0,1	0	0,25	0,5	0	OFF	ON	ON	OFF

Výsledky experimentu č. 10:

Detekce prohrávajících tahů	OFF	ON
# výher	50	50
# playoutů	2332	2435

Efekt optimalizace je prakticky nulový. Pravděpodobně by nebylo dosaženo vyrovnanějšího výsledku ani vzájemnou hrou dvou identických konfigurací.

5.4. Závěr experimentů

I když testování nebylo vyčerpávající, dají se z jeho výsledků odhadnout nejlepší hodnoty pro jednotlivé parametry. Jejich souhrn je následující:

Experiment č.	Parametr	Optimlální hodnota
2	*UTCConst*	0,1
3	Maximální délka playoutu	5
4	*zarazkaConst*	0,25
5	*skoreConst*	0,5
6	Vynechávání uzlů od hloubky	0
7	Chytré simulace	OFF
8	Pamatování překrývajících překážek	ON
9	Pamatování blokujících překážek	ON
10	Detekce prohrávajících tahů	ON

Z parametrů navržených a implementovaných v této práci se jako důležité jeví zejména *UTCConst*, Maximální délka playoutu, *zarazkaConst* a pamatování překrývajících překážek. Zajímavým faktem, který testování ukázalo, je, že je výhodné vynechávat některé uzly už v kořeni, tedy přijmout riziko přehlédnutí dobrého tahu za cenu hlubšího prozkoumání ostatních tahů.

Chytré simulace, heuristika pro zlepšení výběru tahu v průběhu simulace se neukázala jako užitečná. Je však pravděpodobné, že mechanismus výběru tahu v průběhu simulace je možné zvolit tak, aby hru skutečně zlepšoval. Podařilo se to totiž i u mnoha jiných her, často však až po dlouhém experimentování a citlivém vyladění parametrů.

6. Implementace

V této kapitole jsou popsány detaily implementace představeného algoritmu a další důležité věci, které se týkají programu MCTS-Quoridor.

Úmluva pro kapitolu 6: Názvy tříd a funkcí jsou zvýrazněny kurzívou, aby bylo jednodušší odlišit je od stejných výrazů s obvyklým významem.

6.1. Datové struktury

Datové struktury jsou důležitou součástí každého algoritmu. Jejich dobrá či špatná organizace může znamenat řádové rozdíly v časové i paměťové náročnosti. Při návrhu složitějšího programu je vhodné myslet také na jejich přehlednost, robustnost, a snadnou přístupnost pro všechny potřebné funkce.

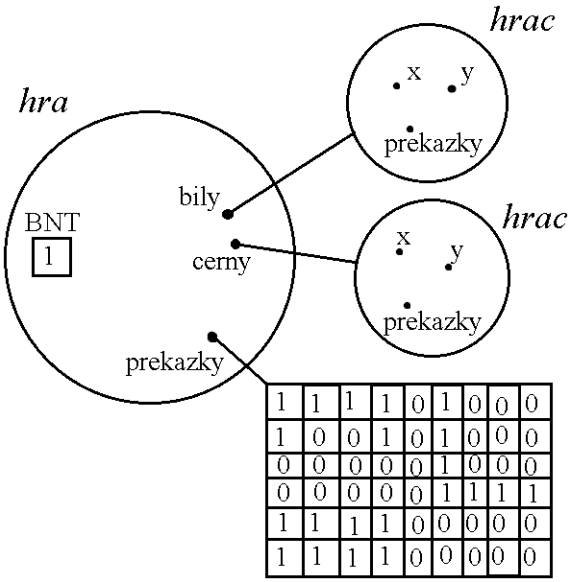
Program MCTS-Quoridor je napsán v programovacím jazyce C#, který je silně objektově zaměřen. Objektově orientované programování proto hraje důležitou roli v návrhu datových struktur.

Jako první představíme reprezentaci pozice ve hře. Za jejím účelem je definována třída *hra*, která nese kompletní informaci o stavu hry. Touto kompletní informací je poloha figurek obou hráčů, počty zbývajících překážek obou hráčů, informace o položených překážkách a nakonec informaci o tom, který hráč je na tahu. Poslední zmíněná informace je na reprezentaci nejjednodušší – stačí jedna proměnná typu boolean.

Zajímavější je reprezentace položených překážek. Ty jsou reprezentovány trojrozměrným polem booleanů, přičemž dva rozměry udávají řádek a sloupec místa pro překážku a třetí rozměr, který může nabývat jen tří hodnot určuje, zda jde o překážku na těchto souřadnicích vertikální, horizontální a nebo křižovatku. Samotná booleovská hodnota na daném indexu jednoduše říká, zda je toto místo obsazeno překážkou, nebo je volné. Výhodou této reprezentace překážek je, že všechny operace prováděné nad polem překážek jsou vykonány v konstantním čase. Jde o zjištění, zda může figurka přejít z jednoho pole na druhé, zjištění, zda je možné

položit překážku na nějaké místo tak, aby se nepřekrývala s jinou a samotné položení překážky. Přitom tato struktura neukládá žádná nepotřebná data navíc.

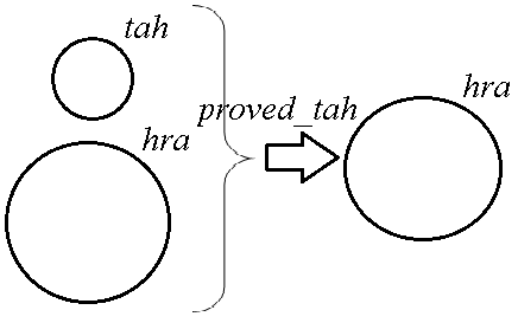
Reprezentace pozice figurky a zbývajících překážek je sloučena do jedné třídy *hrac*, která obsahuje tři integery na tyto informace. Třída *hra* pak obsahuje dva objekty třídy *hrac* – bílého a černého (přesněji obsahuje reference na ně, objekty jsou v C# referenční typy). Touto reprezentací se šetří opakované definování stejných proměnných zvlášť pro bílého a pro černého hráče.



1	1	1	1	0	1	0	0	0
1	0	0	1	0	1	0	0	0
0	0	0	0	0	1	0	0	0
0	0	0	0	0	1	1	1	1
1	1	1	1	0	0	0	0	0
1	1	1	1	0	0	0	0	0

Obr. 6.1: Třída hra

Úpravy na objektu třídy *hra* způsobené průběhem partie se provádějí opět objektem, tentokrát třídy *tah*. Třída tah v sobě nese informaci o tahu, tj. souřadnice a boolean proměnné popisující zda jde o překážku nebo o figurku a zda jde o vertikální nebo horizontální překážku. O přeskládání dat uvnitř objektu *hry* na základě *tahu* se stará metoda *proved_tah*, která bere *hru* a *tah* za parametry. Výhodou této reprezentace informací, je, že objekty *hra* i *tah* mohou být vraceny jako návratové hodnoty funkcí (jazyk C# to umožňuje spolu s používáním objektů jako parametrů funkcí), což je výhodné pro různé funkce které mají za úkol najít další tah.

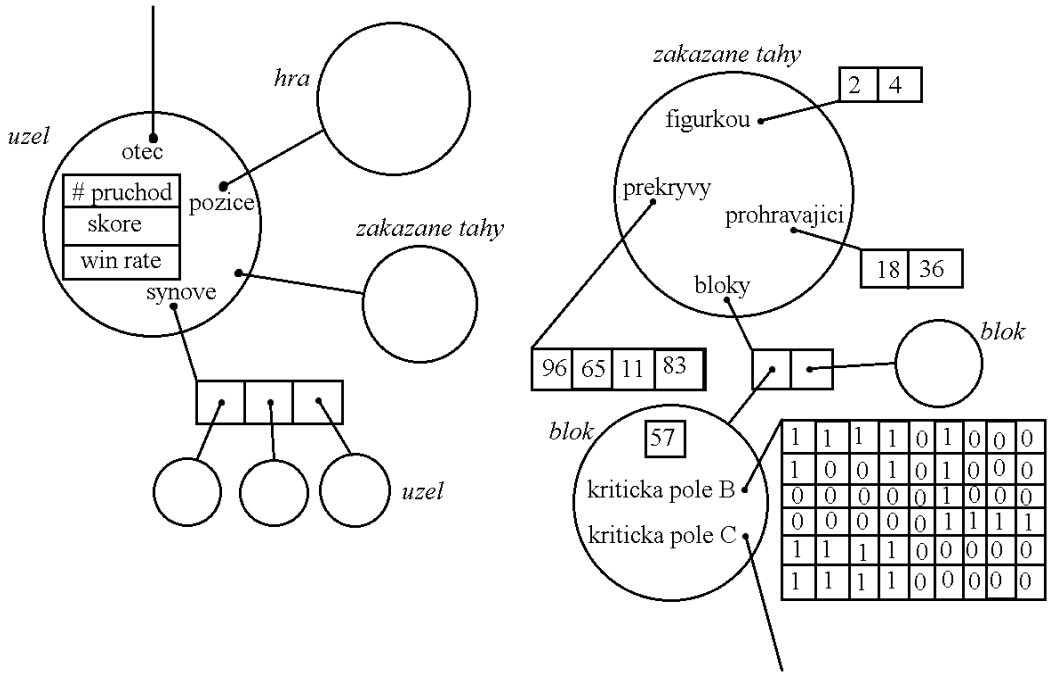


Obr. 6.2: úprava objektu hra objektem tah

Další důležitou datovou strukturou je částečný strom hry, který je tvořen v paměti. Základní stavební jednotkou stromu je třída *uzel*, která nese především jednu pozici, tedy objekt *hry*, potřebné informace pro běh UCT (průchody, skóre, win rate), dále referenci na svého otce ve stromu a seznam synů, který je reprezentován generickým datovým typem *List<uzel>*, jehož výhodou je dynamická velikost. Poslední položkou v uzlu je seznam zakázaných tahů, které jsou uloženy ve vlastní třídě *zakazane_tahy*. Podle konfigurace tak tato vlastnost uzlu může buď zůstat nevyužitá a zabírat jen 4 bajty paměti (pouze reference na objekt), nebo sloužit jako seznam tahů, které je zakázáno losovat při výběru náhodného tahu, aby se ušetřilo zbytečné opakované zjišťování, že je tento tah v dané pozici nemožný.

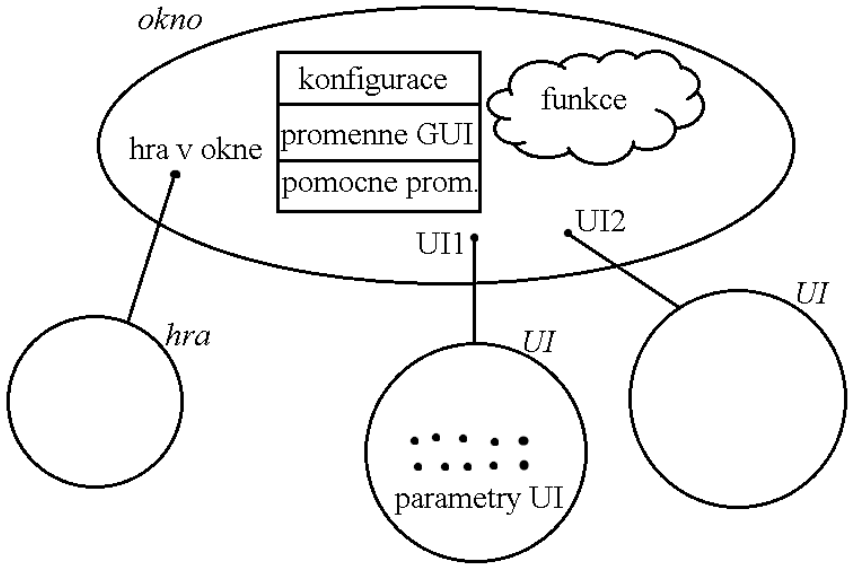
Zakázané tahy se uvnitř třídy ještě rozdělují podle příčiny, z jakého důvodu je tah zakázaný. Toto je nutné kvůli čtení zakázaných tahů uzlem – synem od svého otce. Při eliminaci možných tahů na výběr se totiž berou v potaz pouze ty tahy předků, jejichž ilegalita přetrvává až do konce hry (týká se pokládání překážek, ale ne tahů figurkou). Zakázané tahy se podle příčiny zákazu rozdělují na 4 skupiny: nemožné tahy figurkou, překážky překrývající se s položenou překážkou, překážky odřezávající cestu do cíle a prohrávající tahy. Každá z těchto skupin je reprezentována datovým typem *List<int>*, tedy generickým seznamem celých čísel. Místo celých čísel by bylo možné použít rovnou objekty *tah*, ale tím by silně vzrostla paměťová náročnost stromu. Místo toho se každý tah kóduje celým číslem, přičemž existuje funkce, která umí převádět celá čísla na objekty *tah* a obráceně. Výjimku v této reprezentaci tvoří překážky odřezávající cestu do cíle. Ty nestačí reprezentovat jen celým číslem, ale potřebují ještě seznam kritických polí. Proto jsou tyto tahy reprezentovány vlastním objektem, který obsahuje jednak kód tahu a jednak dvě pole

typu boolean nesoucí informaci, na kterých polích musí stát figurka, aby byl tah ilegální.



Obr. 6.3: struktura uzlu

V jazyce C# jsou funkce definovány pouze jako metody uvnitř tříd. Funkce pro práci s pozicemi a tahy, funkce algoritmu UI i ostatní funkce pro řízení programu jsou metody hlavní třídy *okno*, která má jedinou instanci po celý běh programu. Tato hlavní třída také nese informace o hře, kterou zobrazuje v okně, o aktuálním nastavení daném jednak konfiguračním souborem a jednak uživatelem za běhu aplikace. Poslední nezmíněnou strukturou je třída *UI*, která je kontejnerem parametrů umělé inteligence. Třída *okno* obsahuje dva objekty (reference) třídy *UI*. Tyto označují umělou inteligenci požívanou bílým/černým hráčem a jsou předávány jako parametr funkcím, které s parametry UI pracují.



Obr. 6.4: hlavní třída programu

6.2. Funkce a použité techniky v programu

Funkce v programu lze rozdělit na dva druhy podle účelu: funkce umělé inteligence a obsluha GUI (grafického rozhraní). Zvláštním případem je konstruktor třídy *okno*, který je volán vždy na začátku programu. Konstruktor načítá konfigurační soubor "config.txt", který musí být obsažen ve zdrojovém adresáři programu, a poté inicializuje proměnné *okna* na výchozí hodnoty.

Základní funkcí umělé inteligence je funkce *najdi tah*, která jako vstupní parametr přijímá objekt třídy *hra* a jejím návratovým typem je třída *tah*. Funkce *najdi tah* tvoří z objektů *uzlů* strom hry, který je základem algoritmu UCT. První uzel – kořen stromu je vytvořen na začátku běhu funkce a poté následuje cyklus popsaný v 3.1. Tato funkce také sama rozpozná, které parametry UI (UI1/UI2) má použít podle toho, který hráč je v kořeni na tahu a dalším funkcím už zvolený objekt UI předává zpravidla jako parametr, protože další funkce už nebudou mít k dispozici kořen.

Dalšími důležitými funkcemi jsou *nahodny tah*, *nahodna hra*, *legalni tah* a *hodnota pozice*. Celkově program obsahuje 13 funkcí pro algoritmus UI, 19 pro obsluhu GUI, konstruktor a pomocnou funkci na zpracování konfiguračního souboru.

Funkce *nahodny tah* je založená na vylosování čísla náhodným generátorem. Protože každý tah umíme reprezentovat celým číslem, stačí vylosovat číslo od 0 do počtu všech tahů (na desce 9x9 jich je 132) vylosované číslo však může odpovídat tahu, který v dané situaci není přípustný. Tento fakt zjišťuje booleovská funkce *legalni tah*, která dostává na vstup objekt *hry* a objekt *tahu*.

Funkce *legalni tah* v případě tahu figurkou snadno zkontroluje legalitu tohoto tahu na základě mřížky překážek v příslušné hře. V případě pokládání překážky je rychlé pouze ověření, že se překážka nepřekrývá s jinou, již položenou překážkou. K ověření, zda překážka neodřízne jednomu hráči cestu do cíle je však použito prohledávání do šířky, které musí postupně projít všechna pole desky, má tedy kvadratickou složitost vzhledem k délce strany (je-li deska čtvercová). Z tohoto důvodu způsobuje funkce *legalni tah* malé zdržení při každém jejím použití a je proto snaha využívat ji co nejméně.

Při losování náhodného tahu je proto výhodné nelosovat čísla, u kterých víme, že by funkce *legalni tah* vrátila negativní výsledek. Součástí funkce je proto seznam tahů, které nesmí být vylosovány. Tento seznam je reprezentován strukturou *List<int>* a kódy jednotlivých tahů se v něm mohou objevit jedním z následujících tří způsobů. Za prvé: tah je zaznamenán jako zakázaný v uzlu ze kterého se nový tah hledá, případně v některém z jeho předků. Za druhé: tah již byl v tomto běhu funkce jednou vylosován a neprošel kontrolou funkce *legalni tah*. Za třetí: hledáme nového syna uzlu a na vstupu funkce byl předán seznam již existujících synů. Jejich kódy losovat nechceme, proto jsou do seznamu nežádoucích kódů zařazeny také.

Simulovanou náhodnou hru provádí funkce *nahodna hra*, která dostává na vstupu objekt *uzel* a vrací skóre vypočtené funkcí *hodnota pozice*, což je evaluační funkce popsaná v kapitole 4. Na první pohled se může jevit zbytečné, aby byl v parametru funkce předáván celý *uzel*, když by stačila jen *hra*, která je v něm obsažená. Uzel je však předáván proto, aby optimalizace pamatování zakázaných tahů mohly být použity i v playoutu. Funkce při playoutu opakovaně hledá náhodné tahy funkcí *nahodny tah* a upravuje jimi *hru* v obdrženém *uzlu*. Přitom se stále uchovává a případně rozšiřuje seznam zakázaných tahů, ovšem pouze těch, které jsou překrývajícími, nebo blokujícími překážkami. Jen tyto totiž platí až do konce hry.

Tento cyklus běží, dokud jeden z hráčů v upravované *hře* nedojde do cíle, nebo dokud není naplněn maximální počet tahů v playoutu určený UI aktivního hráče (aktivního v kořeni, nikoliv v upravované hře) .

Tím jsou shrnuty páteřní funkce pro fungování algoritmu MCTS. Funkce na obsluhu GUI jsou technická programátorská záležitost a nebudou zde hlouběji rozebírány.

6.3. Instalace a ovládání programu

Přiložený program MCTS-Quoridor je napsán v jazyce C#. Cílovou platformou je .NET Framework 2.0 a cílovým OS je MS Windows. Pro správný běh programu je nutné, aby byl nainstalovaný .NET Framework 2.0, který je přiložen k programu. Pro instalaci .NET Framework 2.0 je nutné mít nainstalován Windows installer verze 3.1 nebo vyšší, který je rovněž přiložen k programu.

Adresář se spustitelnou kompilací programu obsahuje 4 soubory:

1. spouštěcí soubor "MCTS-Quoridor.exe"
2. konfigurační soubor "config.txt"
3. zdrojový soubor s obrázkem bílé figurky "white.png"
4. zdrojový soubor s obrázkem černé figurky "black.png"

Uvnitř konfiguračního souboru je možné nastavit parametry jednak pro samotnou hru a její pravidla a jednak pro umělou inteligenci zvlášť pro bílého a černého hráče. Všechny parametry se v konfiguračním souboru zapisují ve tvaru [název parametru][mezera][hodnota parametru], každý parametr na nový řádek.

Následuje tabulka všech konfigurovatelných parametrů:

Název	Přípustné hodnoty	Implicit. hodnota	Popis
deska_x	$\mathbb{N}$	9	Šířka hrací plochy
deska_y	$\mathbb{N}$	9	Výška hrací plochy
bily_zarazky	$\mathbb{N}$	10	Počáteční počet překážek bílého
cerny_zarazky	$\mathbb{N}$	10	Počáteční počet překážek černého
bily_start	[A-Z] $\mathbb{N}$	E 1	Pole, na kterém začíná bílá figurka
cerny_start	[A-Z] $\mathbb{N}$	E 9	Pole, na kterém začíná černá figurka
SET	UI1,UI2	-	Začátek deklarací parametrů UI pro bílého (UI1) / černého (UI2) hráče
max_testu	$\mathbb{N}$	10000	Maximální počet playoutů
max_time	$\mathbb{N}$	2000	Maximální čas na přemýšleni
UTCConst	$\mathbb{R}$	0,1	*UTCConst*, viz 3.2
zarazkaConst	$\mathbb{R}$	0,25	*zarazkaConst*, viz 4
skoreConst	$\mathbb{R}$	0,5	*skoreConst*, viz 4
max_delka_simulace	$\mathbb{N}$ nebo -1	5	Maximální počet náhodných tahů v playoutu, -1 značí neomezeně
pamatovat_prekryvajici_prekazky	0/1	1	Přepínač optimalizace pamatování zakázaných tahů
pamatovat_blokujici_prekazky	0/1	1	Přepínač optimalizace pamatování zakázaných tahů
detekce_prohravajicich_tahu	0/1	1	Přepínač optimalizace pamatování zakázaných tahů

vynechavani_uzlu	ℕ nebo -1	-1	Od které úrovně je možné neprocházet všechny syny uzlů, -1 značí nikdy
chytry_playout	0/1	1	Přepínač heuristiky pro chytré playouty

Po spuštění aplikace se otevře okno s hrací plochou a ovládacím panelem. Hra je implicitně nastavena na hru člověk vs. člověk. Ovládání hráče se dá přepínat mezi lidským hráčem a umělou inteligencí na panelu vpravo. Je-li na tahu lidský hráč, provede tah buď kliknutím na pole, na které se chce přesunout, nebo postupným kliknutím na 2 mezery, do kterých chce umístit překážku. Pokud hráč klikne na nedostupné pole, nebo označí nepřípustnou dvojici mezer, tah se neprovede.

Pod ovládáním kontroly jednotlivých hráčů se zobrazují informace o aktuálním stavu hry – o aktivním hráči, o počtu zbývajících překážek každého hráče a historie všech provedených tahů.

V okamžiku, kdy jeden z hráčů dosáhne protější strany hrací plochy, se hra automaticky ukončí a oznámí vítězství daného hráče.

Po ukončení hry se lze vrátit do výchozího stavu stisknutím tlačítka "Nová hra". Toto tlačítko lze použít i kdykoliv během rozehrané hry.

Další možné volby hry jsou:

- Uložení/Nahrání partie, které pro současnou hru vytvoří soubor v adresáři aplikace, který bude možné kdykoliv nahrát a hru tak obnovit.
- Tlačítko Zpět, které po stisknutí vrátí partii o 1 tah zpět.
- Tlačítko Nápověda, které po stisknutí použije mechanismus umělé inteligence k navrhnutí dalšího tahu, tento tah zobrazí na hrací desce a zobrazí zprávu se zápisem tahu, stejným, jako se používá v historii tahů. Po potvrzení zprávy se zobrazený tah na hrací desce opět vrátí.

Poslední funkcí programu je přepínání mezi herním a testovacím módem. Do testovacího módu lze vstoupit kliknutím na tlačítko "Testování". Při vstupu do testovacího módu se zobrazí tři nová tlačítka: Reset, Start a Hrací mód.

Pod tlačítky je číselné počítadlo s šipkami, kterým je možné nastavit požadovaný počet testovacích her. Testování se poté spustí stiskem tlačítka Start. Tím začne série testovacích her, ve kterých proti sobě hrají černý a bílý hráč, každý ovládaný svojí umělou inteligencí nakonfigurovanou v konfiguračním souboru.

V průběhu testování se po každém tahu aktualizují popisky na testovacím panelu, které hlásí, jaký je průběh testování, tj. kolik se již odehrálo her a tahů, počet vítězství obou hráčů a průměrný počet playoutů, který algoritmus provedl na jeden tah, zvlášť pro každého hráče, protože jiné nastavení UI může způsobit jiný počet provedených playoutů.

Po skončení testování se na testovacím panelu objeví hláška, že testování bylo dokončeno a také se změní popis aplikace v liště, aby byl uživatel upozorněn na ukončení testování, pokud má okno aplikace minimalizované do lišty. Před spuštěním dalšího testování je možné vynulovat popisky testování tlačítkem Reset.

Tlačítko Hrací mód přepíná program zpět do hracího módu. Po jeho stisknutí se okno aplikace vrátí do stavu před stisknutím tlačítka Testování.

Závěr

Cílem práce bylo zjistit možnosti aplikace algoritmu MCTS na hru Quoridor. Byla navržena jedna konkrétní aplikace, přičemž v místech, kde bylo možné postupovat v návrhu více různými způsoby, byly tyto způsoby zmíněny. V základních principech algoritmu se práce inspiruje již existujícími aplikacemi MCTS známými z jiných her, ale pro tyto základní principy již navrhuje vlastní metody, mezi kterými navíc dává v několika případech na výběr. Dále práce navrhuje řadu optimalizací a heuristik použitelných v algoritmu.

Přirozeným řešením problému, které konkrétní postupy zvolit, je experimentální srovnání vlastností algoritmu v závislosti na zvolené variantě. Proto byla provedena řada experimentů, které měly rozhodnout, které z navržených postupů jsou lepší a jak optimalizovat parametry algoritmu. Experimenty byly provedeny v testovacím módu přiloženého programu, který je možné používat i pro další výzkum vlastností MCTS aplikovaného na Quoridor.

Testování ukázalo, že je možné zvýšit efektivitu algoritmu vhodnou konfigurací parametrů umělé inteligence. Z navržených optimalizací a heuristik se ukázaly jako více či méně užitečné všechny až na jednu heuristiku (tzv. chytré playouty) a tím byl splněn cíl nalézt doménově specifické varianty MCTS vycházející z vlastností hry Quoridor. Nejlepší nalezené hodnoty jednotlivých parametrů jistě nejsou skutečně nejlepší možné. K jejich dalšímu zlepšování by bylo potřeba provést další experimenty s delšími sériemi testovacích her, experimenty provedené v rámci této práce byly však také poměrně vyčerpávající.

MCTS algoritmy jsou v současné době nejsilnější třídou algoritmů pro počítačové Go a objevují se pokusy přenést tyto algoritmy i na jiné hry. (Chaslot, [4]) Aplikace na hru Quoridor je však novinkou. Quoridor má proti klasickým deskovým hrám, jako jsou šachy nebo Go tu nevýhodu, že není zdaleka tolik teoreticky prozkoumán, a proto jsou v této aplikaci MCTS použity jen povrchní poznatky o hře. Přesto algoritmus dosahuje použitelných výsledků a v případě přidání kvalitních teoretických znalostí o hře Quoridor a dalšího experimentálního odladění parametrů by nabízel ještě velký prostor pro zlepšení.

Seznam použité literatury

[1] M. H. Kalos, P. A. Whitlock: *Monte Carlo Methods*, New York University, 1986

[2] A. Hollosi, M. Pahle: *Sensei's Library*, http://senseis.xmp.net, 2000

[3] R. Coulom: *The Monte Carlo Revolution in Go*, Université Charles de Gaulle, 2009

[4] G. Chaslot, S. Bakkes, I. Szita, P. Spronck: *Monte-Carlo Tree Search: A New Framework for Game AI*, Universiteit Maastricht, Netherlands, 2008

[5] S. Gelly, Y. Wang, R. Munos, O. Teytaud: *Modification of UCT with patterns in Monte-Carlo Go*, Institut National de Recherche en Informatique et en Automatique, 2006

[6] S. Gelly, D. Silber: *Combining online and offline knowledge in UCT*, Univ. Paris Sud, France, University of Alberta, Edmonton, 2008

[7] L. Glendenning: *Mastering Quoridor*, University of New Mexico, 2002

Seznam použitých zkratek

MC – Monte Carlo – *označení pro širokou třídu metod založených na provádění množství náhodných pokusů*
MCTS – Monte Carlo Tree Search – *algoritmus na prohledávání stromu hry založený na metodě Monte Carlo*
UCT – Upper Confidence bounds applied for Trees – *mechanismus používaný v MCTS, vynalezený v roce 2006, je odvozen od podobného mechanismu UCB (Upper Confidence Bounds)*

ICGA - International Computer Game Association - *Mezinárodní Asociace pro Počítačové hry, založena r. 1977 programátory šachových programů za účelem pořádání soutěží v umělé inteligenci pro hry*

UI – umělá inteligence – *zkratka může značit i uživatelské rozhraní, v kontextu této práce však vždy značí umělou inteligenci*

GUI – Graphical User Interface – *grafické uživatelské rozhraní, část programu určená ke komunikaci s uživatelem, typicky tlačítka, okna apod.*

Přílohy

[1] Program MCTS-Quoridor – Program je obsažen na přiloženém CD a to jak v podobě spustitelné kompilace, tak v podobě zdrojových kódů. K programu jsou přiloženy potřebné instalační soubory pro .NET Framework 2.0 a Windows Installer.