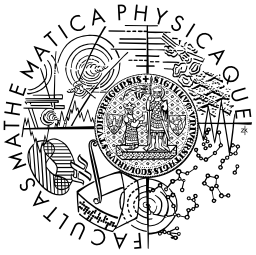


MATEMATICKO-FYZIKÁLNÍ FAKULTA
Univerzita Karlova

BAKALÁŘSKÁ PRÁCE

Jan Palášek

Umělá inteligence pro deskovou hru Warlight

Katedra softwaru a výuky informatiky

Vedoucí bakalářské práce: Mgr. Jakub Gemrot, Ph.D.

Studijní program: Informatika

Studijní obor: Softwarové a datové inženýrství

Praha 2019

Prohlašuji, že jsem tuto bakalářskou práci vypracoval(a) 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 má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V ..... dne ..... Podpis autora

Na tomto místě bych rád v první řadě poděkoval vedoucímu práce Mgr. Jakubu Gemrotovi, Ph.D. za jeho zájem o práci a cenné rady. Dále bych chtěl poděkovat své přítelkyni, rodičům a sestře za podporu nejen při psaní bakalářské práce, ale i během studia.

Název práce: Umělá inteligence pro deskovou hru Warlight

Autor: Jan Palášek

Katedra: Katedra softwaru a výuky informatiky

Vedoucí bakalářské práce: Mgr. Jakub Gemrot, Ph.D., Katedra softwaru a výuky informatiky

Abstrakt: Warlight, inspirovaný deskovou hrou Risk, představuje výzvu pro tvorbu umělé inteligence z důvodu velké výpočetní složitosti.

Práce implementuje umělou inteligenci do této hry. Součástí je také simulátor, možnost hry proti AI i proti jinému lidskému hráči ve formě hotseat (multiplayer hry na jednom počítači). Práce je navržena tak, aby umožnila použití tohoto frameworku pro další vývoj a testování umělé inteligence.

Klíčová slova: umělá inteligence, hry dvou hráčů, desková hra, Warlight

Title: Artificial Intelligence for Warlight Board Game

Author: Jan Palášek

Department: Department of Software Engineering

Supervisor: Mgr. Jakub Gemrot, Ph.D., Department of Software Engineering

Abstract: Warlight, inspired by the board game Risk, represents a challenge for making an artificial player due to its big complexity.

This thesis implements an artificial player for this game. Part of this thesis is a simulator, possibility to play against an AI player and a possibility to play against another human in the form of a hotseat game (multiplayer game on one computer). A framework was designed in order to be used for future development and testing of AI in this game.

Keywords: artificial intelligence, two-player games, board game, Warlight

Obsah

Úvod

0.1 Základní informace o hře

Warlight, inspirovaný deskovou hrou Risk, je hra pro více hráčů zjednodušeně simulující skutečný válečný konflikt. Každý hráč začíná na dvou územích. Cílem hry je dobýt všechna území vlastněná ostatními hráči. Nedílnou součástí této hry je také náhoda, která rozhoduje o množství ztrát při každém boji.

Terminologická poznámka Následující seznam pojmů definuje synonyma používaná v práci. Nejprve je nadefinován přesný pojem a poté seznam jeho používaných synonym:

- region - území,
- armáda - jednotky(a),
- umělá inteligence - AI, bot.

0.2 Motivace

I přes existenci soutěže vypsané Riddles.io [1] je oblast tvorby umělé inteligence pro hru Warlight nepříliš zmapovaná. Důvodem je nezveřejňování existujících implementací, nebo jejich malá či vůbec žádná dokumentace. Neprobádanost, spolu s velkým větvícím faktorem hry, nás motivuje k pokusu o vytvoření umělé inteligence.

0.3 Cíl práce

Cílem práce je naimplementovat umělou inteligenci do hry Warlight. Pro snadnější vývoj umělé inteligence je dílčím cílem práce přidat simulátor pro pozorování her botů proti sobě a hru ve formě singleplayer nebo hotseat multiplayer hry (více hráčů na jednom počítači).

0.4 Změna v zadání

Jako součástí práce jsme chtěli změřit síly naší umělé inteligence s ostatními v soutěži vypsané Riddles.io. Tato snaha byla opuštěna z důvodu nefunkční kompilační fronty na jejich serveru a neodpovídání na naši snahu o kontakt.

0.5 Související práce

GG [4] a Benjamin628 [2] ve svých pracích popisují, jakými pravidly by se měl lidský hráč řídit při hraní Warlightu. Poznatky z těchto prací jsou využity při tvorbě generátoru akcí pro AI.

[1] `http://theaigames.com/competitions/warlight-ai-challenge`

Norman [6] dodává sadu rad, jak by se umělá inteligence měla chovat v různých herních situacích. To je využito při implementaci generátoru akcí a funkcí ohodnocujících stav v AI.

Chaslot a kol. [3] rozebírají přístupy k implementaci algoritmu Monte Carlo tree search paralelně. Jejich efektivita je měřena na hře Go. Prací zmíněná tzv. kořenová paralelizace je použita pro paralelizování výpočtu v naší implementaci umělé inteligence.

0.6 Struktura práce

Práce se skládá ze tří kapitol vyjma úvodu a závěru:

- Pravidla hry - cílem první kapitoly je detailně popsat čtenáři pravidla hry,
- Analýza - v druhé kapitole zanalyzujeme problém tvorby umělé inteligence do hry Warlight a vybereme vhodný algoritmus pro její realizaci,
- Umělá inteligence - ve třetí kapitole popíšeme použitý algoritmus a jeho modifikace přizpůsobené znalostem této hry. Dále zanalyzujeme objektový návrh a popíšeme jeho hlavní komponenty. Nakonec provedeme experimenty a rozebereme jejich výsledky. Zaměříme se na průzkum nedostatků a problémová místa.

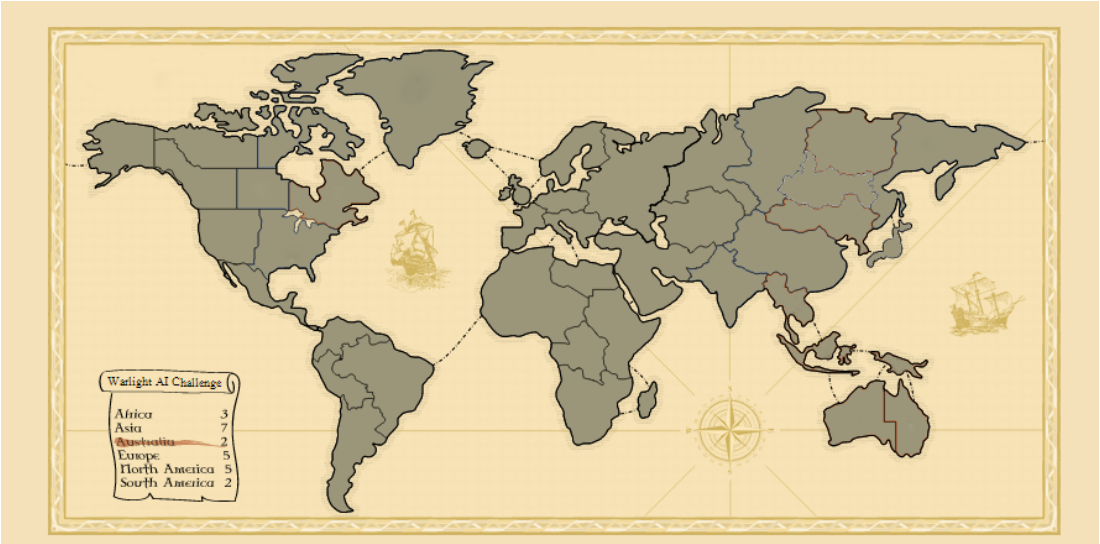
1. Pravidla hry

Pravidla hry Warlight jsou relativně volná, hra se dá hrát na spoustu různých nastavení, která se liší především v míře náhody při útočení a způsobu volení regionů na začátku hry. Cílem této kapitoly je popsat pravidla hry Warlight s nastavením implementovaným v této práci.

1.1 Mapa

Hra se odehrává na mapě. Ta se dělí na regiony, nejmenší územní celky této hry. Každý region má armádu, seznam sousedních regionů, a buď hráče, který ho vlastní, nebo je neobsazený. Regiony se dále shlukují do větších územních celků, super regionů. Mapou může být libovolný neorientovaný graf regionů, kde vrcholy jsou regiony a hrany jsou mezi vrcholy takovými, že reprezentují regiony, které spolu sousedí.

V práci je naimplementována jediná mapa sloužící pro účely testování a hraní – mapa světa.



Obrázek 1.1: Mapa světa [7].

Ohraničená území představují regiony, super regiony jsou kontinenty – Afrika, Asie, Austrálie, Evropa a Severní a Jižní Amerika.

1.2 Začátek hry

Na začátku hry je pro každého hráče vygenerována množina regionů tak, že od každého super regionu je zvolen právě jeden. Hráč si z této množiny vybírá dva regiony. Tato území představují výchozí body, ze kterých bude obsazovat další. Commitem potvrzuje své předchozí akce.

1.3 Průběh hry

Hra se dělí na herní kola. Každé z nich se skládá z tahů, kde každý hráč přispívá do kola právě jedním tahem.

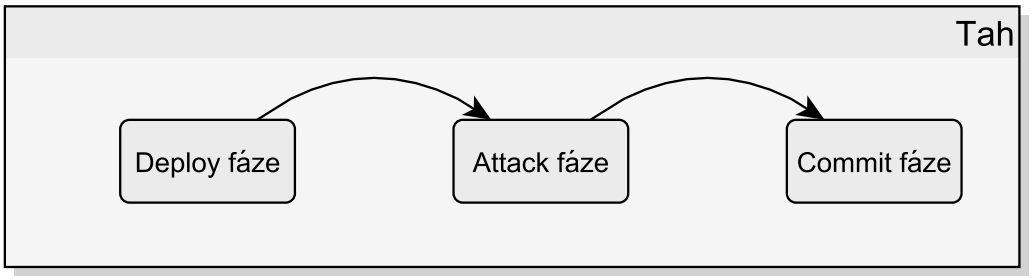
Během hry se hráči střídají po tazích. Odehrají-li všichni hráči své tahy, dojde k ukončení kola. Poté dojde k výpočtu nového stavu hry, tedy výpočtu ztrát jednotek a případných změn vlastníků regionů a k započetí nového kola. Tyto kroky se opakují, dokud jeden hráč neobsadí regiony všech ostatních hráčů, a nevyhraje tak hru.

1.4 Tah

Každý hráč přispívá do kola právě jedním tahem. Jakmile všichni ukončí své tahy commitem, spustí se výpočet kola, který aktualizuje herní stav. Tahy posledního kola se nejprve zlinearizují, poté následuje výpočet změn.

Tah se dělí na tři fáze: deploy, attack a commit. V deploy fázi hráč staví armádu, v attack fázi posílá útoky a v commit fázi potvrzuje své předchozí akce.

Po odehrání deploy fáze hráč přechází do attack fáze, po jejímž dokončení přechází do commit fáze. Svůj tah poté hráč dokončuje potvrzením svých předchozích akcí v commit fázi.



Obrázek 1.2: Přechody mezi fázemi.

1.4.1 Deploy fáze

V této fázi hráč staví armádu na jeho regiony.

Deploy akcí nazveme jev, kdy hráč postaví nenulový počet jednotek na daný region. Každá deploy fáze se skládá z 0 nebo více deploy akcí.

Hráč má určený maximální počet jednotek, které může v daném tahu postavit. Na začátku hry může stavět 5 jednotek. Dobude-li nějaký super region, zvýší se mu přísun jednotek o bonus definovaný super regionem. Pokud o super region přijde, přijde také o bonus jím poskytovaný. Super regiony mapy světa mají následující bonusy:

- Asie - 7
- Evropa - 5
- Severní Amerika - 5

- Jižní Amerika - 2
- Afrika - 3
- Austrálie - 2

1.4.2 Attack fáze

V této fázi hráč útočí armádou vždy ze svého regionu na region sousední, popřípadě jednotky přesouvá mezi svými sousedními regiony.

Attack akcí nazveme jev, kdy hráč pošle nenulový počet jednotek z jím vlastněného regionu na region sousední. Attack fáze se skládá z nuly nebo více attack akcí. Pokud v jedné attack fázi je více attack akcí takových, že vychází a míří do stejných regionů, pak jsou tyto akce sloučeny – z více attack akcí se vytvoří jedna, která bude vycházet a mířit do stejných regionů, a velikost její armády bude součet všech vyslaných jednotek sloučených akcí.

Při útoku nelze útočit s celou armádou. Na regionu musí zůstat alespoň jedna jednotka.

1.4.3 Commit fáze

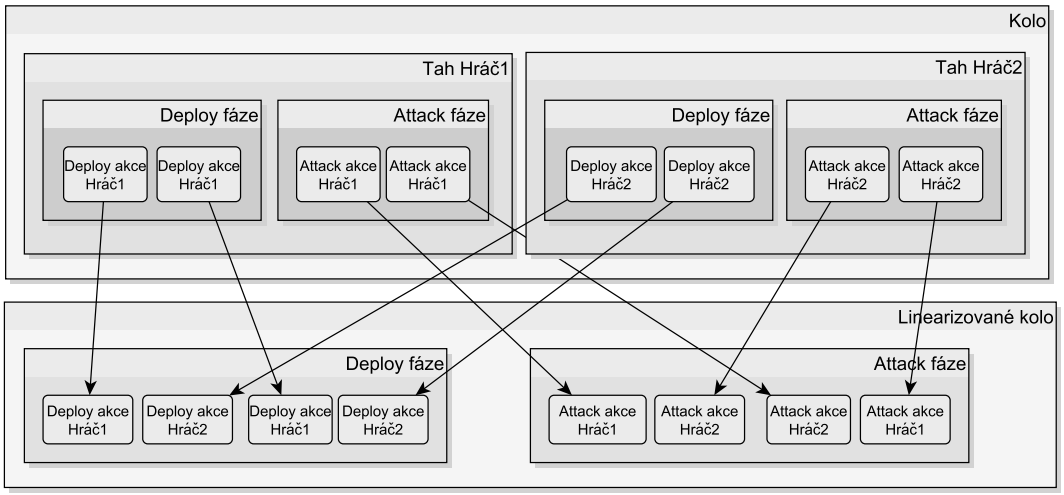
V této fázi hráč potvrzuje akce, které provedl v deploy a attack fázi. Po tomto potvrzení již není možné je vrátit a hráčův tah je považován za uzavřený.

1.5 Kolo

Každý hráč přispívá do kola právě jedním tahem. Jakmile všichni ukončí své tahy commitem, spustí se výpočet kola, který aktualizuje herní stav. Tahy posledního kola se nejprve zlinearizují, poté následuje výpočet změn.

1.5.1 Linearizace

Algoritmus nejprve zlinearizuje deploy akce tak, že pro každý index `i=1,...,max(#deployAkcíLibovolnéhoTahu)` vezme i-té deploy akce všech tahů daného kola (v libovolném pořadí) a přidá je do výstupního seznamu. Poté zlinearizuje attack akce tak, že pro každý index `i=1,...,max(#attackAkcíLibovolnéhoTahu)` vezme i-té attack akce každého tahu, a v náhodném pořadí je přidá do výstupního seznamu. Linearizované kolo je tvořeno dvěma výše popsanými výstupními seznamy.



Obrázek 1.3: Linearizace.

```
Linearizuj() : kolo
    tahy = { všechny t | t je odehraný tah }
    zpřeházejNáhodně(tahy);

    // zlinearizuj deploy akce
    deploy = {}
    pro každý index i = 1, ...,
        maximum(počet deploy akcí libovolného tahu)
        iDeploy := { i-té deploy akce všech tahů }
        deploy.přidej(iDeploy)

    // zlinearizuj attack akce
    attack = {}
    pro každý index i = 1, ...,
        maximum(počet attack akcí libovolného tahu)
        iAttack := { i-té attack akce všech tahů }
        zpřeházejNáhodně(iAttack)
        deploy.přidej(iAttack)

    linearizovanéKolo = (deploy, attack)
    vrať linearizovanéKolo
```

Algoritmus 1.1: Linearizace kola

1.5.2 Výpočet změn kola

Při výpočtu změn kola dojde k aktualizaci herního stavu provedením všech zlinearizovaných akcí.

Nejprve jsou spuštěny všechny deploy akce – jednotky jsou přidány na regiony hráčů.

Poté jsou spuštěny všechny attack akce – dojde k výpočtu ztrát jednotek v boji a případné změně vlastníků regionů. Výpočet ztrát se řídí následujícími pravidly:

- Každá útočící jednotka má 60% šanci na zabití bránící jednotky,
- každá bránící jednotka má 70% šanci na zabití útočící jednotky.

Následující algoritmus spočítá všechny změny způsobené útoky a zaktualizuje tak současný herní stav. Algoritmus implementuje následující pravidla:

- Pokud útočící region změnil vlastníka, tento útok se neprovede (byl by proveden jednotkami hráče, který útok neposlal),
- nelze zaútočit tak, že na regionu nezůstane žádná jednotka – vždy musí zůstat alespoň 1,
- útočí-li hráč na svůj region, nedochází ke ztrátám na jednotkách,
- pokud při útoku dojde k zabití útočících i bránících jednotek, na bránící region je přiřazena 1 jednotka (a nemění se jeho majitel),
- pokud při útoku přežily bránící i útočící jednotky, zbytek přeživších útočících jednotek se vrátí na region, ze kterého přišly.

```
spočítejAttacky(linearizovanéAttacky)
    pro každý attack v linearizovanéAttacky
        X := attack.útočícíRegion
        Y := attack.bráníRegion
        útočícíHráč := attack.útočícíHráč;

        // útočící region změnil vlastníka
        pokud X.vlastník != útočícíHráč
            přeskoč tento útok

        // vždy musí zbýt alespoň jedna
        // jednotka na regionu
        reálnáÚtočícíArmáda := minimum(
            attack.útočícíArmáda, X.armáda - 1)
        bráníArmáda := Y.armáda

        // hráč útočí na svůj region
        pokud útočícíHráč == Y.vlastník
            přesuň jednotky
        jinak
            // spočítej zabité jednotky
            zabitéÚtočícíJednotky :=
                spočítejZabitéÚtočícíJednotky(
                    reálnáÚtočícíArmáda, bráníArmáda)
```

```
zabitéBráníciJednotky :=
    spočítejZabitéBráníciJednotky(
        bráníciArmáda, reálnáÚtočícíArmáda)

pokud byly zabity všechny útočící
i bránící jednotky
    bráníciArmáda := 1
jinak pokud byly zabity všechny
bránící, ale útočící ne
    Y.vlastník := útočícíHráč
jinak pokud přežily i bránící i útočící
    vrať se s přeživšími útočícími
    jednotkami zpět na X
jinak pokud přežily bránící
    // nic nedělej
```

Algoritmus 1.2: Spočítej útoky

2. Analýza

V této kapitole zanalyzujeme problematiku tvorby umělé inteligence do hry Warlight a určíme vhodný přístup k její implementaci.

Terminologie

- náš hráč - hráč, z jehož pohledu se snažíme najít nejlepší tah,
- nepřátelský hráč - hráč, který není náš hráč.

2.1 Složitost hry

Problémem Warlightu je, že má velmi velký herní strom. Jeho větvení je určené počtem možných tahů hráče.

Tah se skládá z deploy a attack akcí. Na pořadí deploy akcí jednotlivých tahů nezáleží, pořadí attack akcí však dokáže znatelně ovlivnit průběh kola. Předpokládejme, že hráč si může na začátku tahu postavit k jednotek a vlastní n regionů. Potom počet způsobů, jak může postavit k jednotek na n regionů (z kombinací s opakováním) je:

$$\binom{n+k-1}{k-1}$$

Zaměřme se nyní na attack fázi. Nechť s je průměrný počet sousedů regionů. Potom z jednoho regionu můžeme zaútočit na jeho sousedy $\sum_{i=0}^{s} \binom{s}{i} \cdot i!$ způsoby, protože máme celkově až s možností, jak provést útok, a rozhodujeme se, kterou z nich vybereme. Protože u útoku záleží na pořadí, počítáme tak vlastně uspořádaný počet podmnožin.

Celkový počet možností, kolika můžeme zaútočit z našich n regionů, je

$$\left(\sum_{i=0}^{s} \binom{s}{i} \cdot i!\right)^n$$

, protože stačí situaci pro jeden region výše spočítanou opakovat pro všechny námi vlastněné regiony (ve skutečnosti umocnění na n-tou nezapočítáváme závislost na pořadí útoků z jednotlivých regionů, ale námi uvedený výpočet jako dolní odhad stačí). Celkový počet odehrání tahu se skládá z kartézského součinu všech možností odehrání deploy fáze a attack fáze, a je tedy

$$\binom{n+k-1}{k-1} \cdot \left(\sum_{i=0}^{s} \binom{s}{i} \cdot i!\right)^n$$

.

Zkusme nyní dosadit hodnoty tak, aby odpovídaly běžné situaci ve Warlightu na mapě světa. Berme, že průměrný počet sousedů regionů je 3 a mějme 8 regionů s možností postavit si 7 jednotek v rámci tohoto kola. Potom počet způsobů odehrání jednoho tahu je

$$\binom{8+7-1}{7-1} \cdot \left(\sum_{i=0}^{3} \left[\binom{3}{i} \cdot i!\right]\right)^{8} \approx 10^{13}$$

. Běžná hra má v průměru přibližně 30 kol, tedy 60 tahů při hře dvou hráčů. Z toho plyne, že složitost takovéto hry Warlight je $(10^{13})^{60} = 10^{780}$. Následující tabulka porovnává Warlight pro dva hráče se složitostí ostatních her.

Název hry	**Větvící faktor**	**Počet tahů**	**Složitost hry**
Tic-tac-toe	4	9	10^5
Gomoku	230	30	10^{71}
Šachy	35	70	10^{110}
Warlight (2 hráči)	10^{13}	60	10^{780}

Tabulka 2.1: Porovnání složitostí her

Pozn.: Jedním tahem se myslí série akcí jednoho hráče, kterou odehraje předtím, než začne hráč další hráč.

Další výzvou je nedeterminismus útoku. Jednotky mají procentuální šanci na zabití, tedy výsledek každého souboje je do jisté míry randomizující prvek výpočtu. To by v určitých chvílích mohlo hrát v náš neprospěch.

Vezměme si situaci na obrázku níže.

Obrázek 2.1: Náhodnost ve Warlightu

Náš hráč je žlutý a je ohrožen nepřátelským hráčem. Našim cílem v této situaci je ubránit super region Jižní Amerika. Díky náhodnosti by však boj mezi našimi armádami mohl dopadnout různě. S tím je potřeba počítat a pojistit se například postavením větší armády na bránící region.

Pokud bychom si situaci zjednodušili a rozvíjeli pouze pro nás nejhorší, středně dobrou a nejlepší pozici, větvící faktor by se nám zvětšil na 10^{39} a složitost hry na 10^{2340}.

Potřebujeme zakomponovat tuto náhodu do algoritmu tak, aby nezvyšovala již tak dost velkou složitost hry. Je zřejmé, že nemůžeme prozkoumávat všechny možnosti a je potřeba zavést heuristiky na prozkoumávání pouze kvalitnějších tahů.

2.2 Časové omezení

Časová doba výpočtu je značně omezená, protože hráč musí přijít s odpovědí do několika sekund. Maximální doba výpočtu by také ideálně měla být nastavitelná. Pokud hráč hraje s umělou inteligencí na lehkou obtížnost, očekává, že počítači bude trvat vymyslet lepší tah kratší dobu než například na obtížnost těžkou.

2.3 Volba algoritmu umělé inteligence

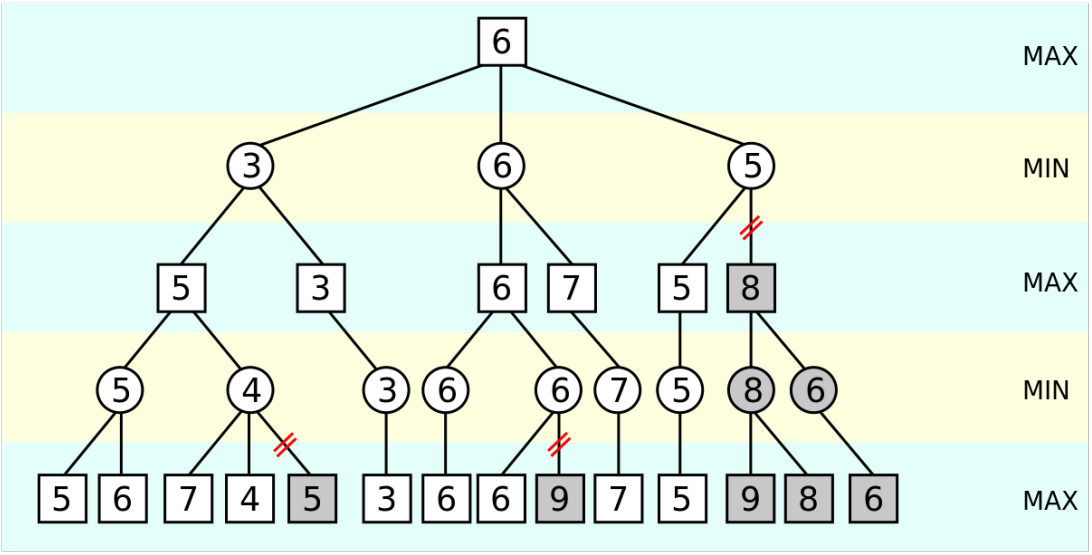
Složitost hry a časové omezení klade náročné podmínky na volbu algoritmu pro umělou inteligenci. Pro práci byly zvažovány dva – minimax a Monte Carlo tree search.

2.3.1 Minimax

Minimax je algoritmus používaný pro hry dvou hráčů. Může stavět strom hry, kde v každém vrcholu je uloženo číslo reprezentující kvalitu pozice, dá se však také naimplementovat pomocí backtrackingu použitím konstantní paměti. Kvalita pozice je určena ohodnocovací funkcí, která přiřazuje pozici číslo. První hráč se pokouší toto číslo maximalizovat, zatímco ten druhý se ho pokouší minimalizovat. Pro efektivní výpočet je potřeba, aby ohodnocovací funkce byla velmi rychlá, protože se používá často.

Alfa-beta ořezávání

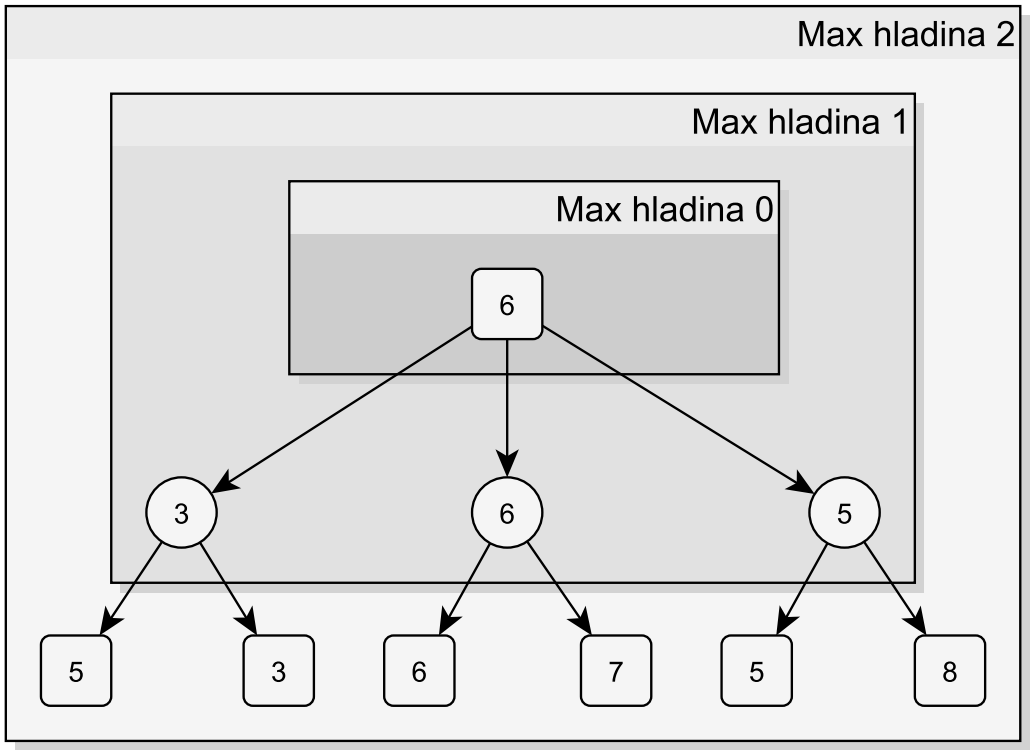
Alfa-beta ořezávání je optimalizace minimaxu umožňující v jisté fázi výpočtu určit, které větve stromu už není třeba prozkoumávat, a odříznout je. K této situaci dojde, když algoritmus zjistí, že by nikdy pro hráče nebylo výhodné zvolit tah reprezentovaný touto větví. Z toho plyne, že pokud budeme nejprve prozkoumávat kvalitní tahy, umožní nám to oříznout spoustu větví.



Obrázek 2.2: Alfa-beta ořezávání [8]

Strom je vytvářen rekurzivně zleva doprava. MIN/MAX na hladině stromu znamená, že v dané úrovni hráč minimalizuje/maximalizuje ohodnocení. Vezmeme-li si případ nejdále vpravo, pak vidíme, že minimalizující hráč zvolil pokračování s hodnotou 5, a tudíž když narazí na pokračování s hodnotou 8, tak je zřejmé, že to pro něj není výhodné, a tedy tato větev se dá odříznout a nemusí se vůbec počítat.

Aby ohodnocení bylo dostatečně přesné, musí být strom nejen prozkoumán do dostatečné hloubky, ale i do šířky. Běžný minimax (případně s alfa-beta ořezáváním) by nejprve prozkoumal až do požadované hloubky jednu větev, a až potom by přešel na druhou. To by mohl být problém v případě značně omezeného času. Proto bychom mohli použít metodu *iterativního prohlubování*. Jak již plyne z názvu, strom hry je iterativně prohlubován po hladinách. Nejprve je prozkoumána hladina 0, pak hladiny 0 1, pak hladiny 0 1 2,... Technika garantuje, že prohledání n-té hladiny je stejně náročné jako prohledání hladin od 0-té až do $n-1$-ní, díky čemuž tato technika je nejhůře dvakrát horší než optimum. Na rozdíl od základního minimaxu však dokáže garantovat vrácení nejlepšího dosud nalezeného tahu po požadovaném časovém úseku, předpokládáme-li že dokončí alespoň první hladinu.



Obrázek 2.3: Iterativní prohlubování

Algoritmus nejprve prozkoumá hladinu 0, poté hladiny 0 1, pak hladiny 0 1 2.

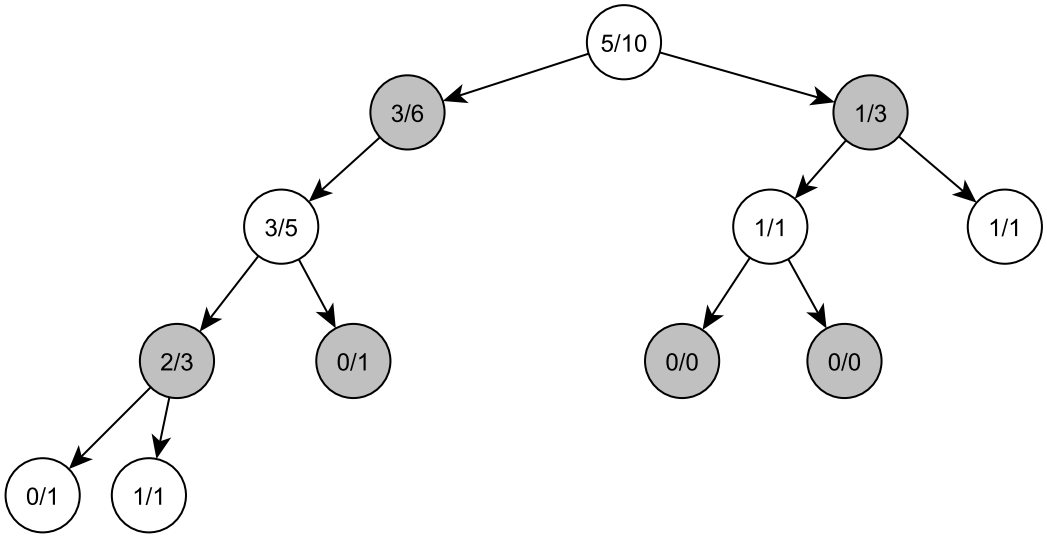
Ačkoliv minimax s výše uvedenými optimalizacemi garantuje vrácení dosud nejlepšího tahu v prakticky libovolný časový okamžik, technika se nezaměřuje na prohledávání nejlepších možností. Kvůli tomu ve hrách s větším větvícím faktorem bude potenciálně ztrácet hodně času s neperspektivními pokračováními.

2.3.2 Monte Carlo tree search

Monte Carlo tree search (zkráceně MCTS) je algoritmus, jehož cílem je najít nejlepší tah pro daný stav hry. Pro tento účel je stavěn strom hry.

Strom hry Strom hry je strom, jehož vrcholy představují stavy hry a hrany představují tahy. Ve vrcholu je navíc uložen počet výher a počet her jeho stavu hry.

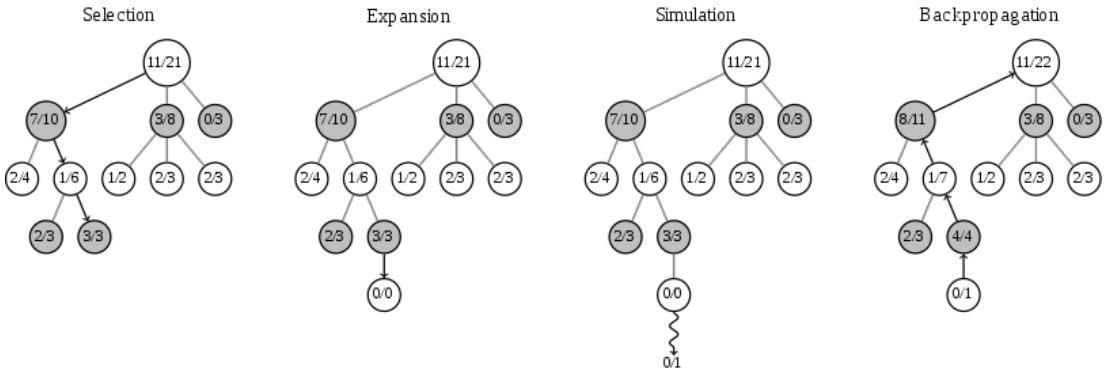
Každý vrchol vždy patří určitému hráči. To, komu patří, se střídá po hladinách stromu: liché patří našemu hráči, sudé nepřátelskému hráči. Hrany patří majiteli vrcholu, ze kterého míří. V kořeni je uložen stav hry, ze kterého se pokoušíme nalézt nejlepší tah. Nejlepší tah pro každý vrchol je vždy reprezentován takovou hranou, která vede z něj do vrcholu s nejvyšším počtem her.



Obrázek 2.4: Strom hry

Bílé vrcholy jsou vrcholy našeho hráče, šedé vrcholy jsou hráče nepřátelského. Čísla vepsaná ve vrcholech *m/n* představují *početVýher/početHer* daného vrcholu.

Algoritmus popisují čtyři fáze: selekce, expanze, simulace a zpětná propagace. Ty jsou v tomto pořadí opakovány až do přerušení běhu. Následují sekce jsou zasvěcené jejich popsáním.



Obrázek 2.5: Monte Carlo tree search fáze[9]

Selekce

Úlohou selekce je najít list stromu, jehož stav bude algoritmus dále prozkoumávat.

```
selekcePotomka(kořen) : vrchol
    dokud vrchol != list opakuj
        vrchol := zvolVhodnéhoPotomka(vrchol.potomci);
    vrať vrchol;
```

Algoritmus 2.1: Selekce

Ačkoliv hlavní myšlenkou algoritmu je rozvíjet tahy, které se jeví být nejlepšími, je potřeba také rozvíjet málo prozkoumané tahy. Kdybychom neprohledávali nové varianty, mohli bychom vynechat tah, který se zprvu jevil jako špatný, ale ve výsledku by byl nejlepším tahem v dané pozici.

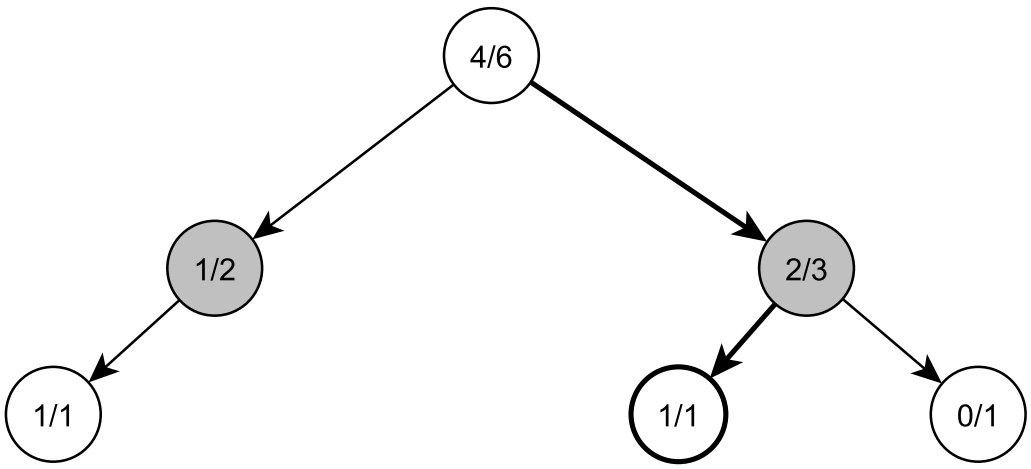
Kocsis a Szepesvári [5] navrhli funkci na volení vhodného potomka při sestupu do listu – *UCT* (Upper Confidence Bound 1 applied to trees). Její vzorec je

$$\frac{w_i}{n_i} + c \cdot \sqrt{\frac{\ln(N_i)}{n_i}}$$

, kde

- w_i - počet výher v daném vrcholu,
- n_i - počet her v daném vrcholu,
- N_i - počet her v rodiči daného vrcholu,
- c - konstanta, teoreticky rovná $\sqrt{2}$, typicky volena empiricky.

Pokud n_i je rovno 0, pak hodnota tohoto vrcholu je ∞. V takové situaci je tento vrchol při příštím sestupu do listu zvolen přednostně.



Obrázek 2.6: Selekce

Tučně zvýrazněné šipky ukazují hrany, po kterých selekce sestupuje z kořene do listu. Tučně zvýrazněný vrchol je list, který byl selekcí zvolen.

Expanze

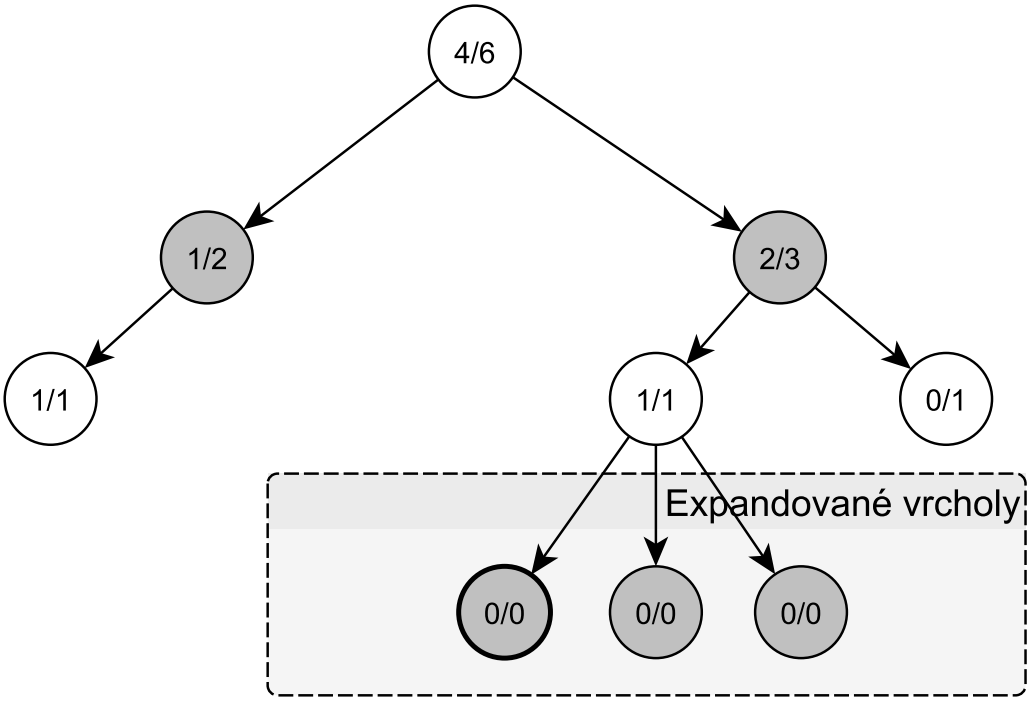
Účelem expanze je přidat stavy (nebo stav), které se budou dále prozkoumávat a vybrat jeden z nich, který se bude rozvíjet ve fázi simulace.

Algoritmus níže expanduje vrchol *list* v parametru tak, že mu přidá potomky a prvního z nich vrátí.

```
expanduj(list)
    // přidej potomky listu
```

```
přidejPotomky(list);

// vrať prvního potomka
vrať list[1];
```

Algoritmus 2.2: Expanze



Obrázek 2.7: Expanze

Bílé vrcholy představují vrcholy našeho hráče, šedé nepřátelského hráče. Vrchol označený tučně zvýrazněným rámečkem je vrácen expanzí.

Simulace

V této fázi je ohodnocen expanzí vybraný vrchol. Toto ohodnocení je získáno simulací neboli odehráváním tahů.

Simulaci dělíme na dva typy podle toho, jakým způsobem je vedeno odehrávání tahů – lehkou a těžkou.

Lehká simulace Hlavní myšlenkou lehké simulace (light playout) je, že odehrajeme-li náhodně dostatečný počet her, ze kterých vždy získáme výsledek výhra/prohra, dokážeme tímto způsobem ohodnotit kvalitu tahu, ze kterého jsme náhodná odehrání vedli. Výhodou je, že pro aplikaci lehké simulace není potřeba znát žádné detaily hry, ale stačí pouze dokázat určit, která strana vyhrála. Vyhraje-li náš hráč, simulace vrací hodnotu 1, jinak vrací hodnotu 0.

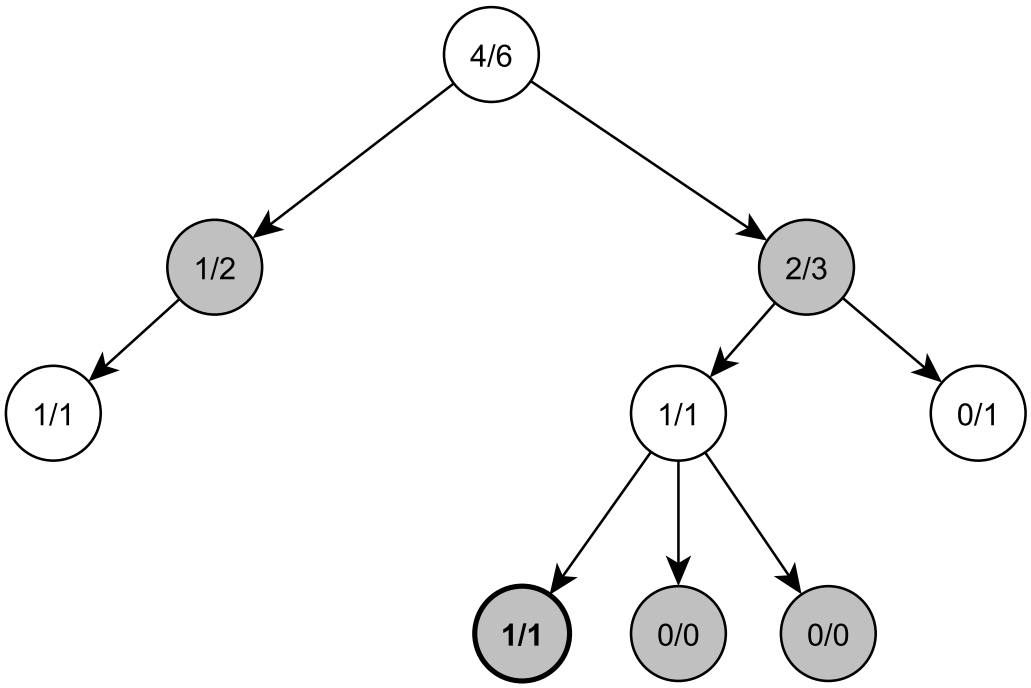
Těžká simulace Těžká simulace (heavy playout) se snaží místo náhodného odehrávání volit tahy, které se vyplatí prozkoumávat. Využívá k tomu znalost

konkrétního problému, tedy např. znalost hry Warlight. Díky tomu má výsledek takové simulace mnohem větší váhu. Definovat takový tah je ale obtížné, nalézt ho je výpočetně náročné a vyžaduje dobrou znalost hry.

Algoritmus níže popisuje fázi simulace. Je proveden playout, který jako výsledek vrátí, zdali vyhrál náš hráč nebo nepřátelský. Toto číslo je vypropagováno ze simulace ven.

```
Simuluj(list) : celé číslo
    // proveď simulaci, vrať její výsledek
    // 1 = náš hráč vyhrál, 0 = nepřátelský hráč vyhrál
    výsledekSimulace := udělejPlayout(list.stav);

    vrať výsledekSimulace;
```

Algoritmus 2.3: Simulace

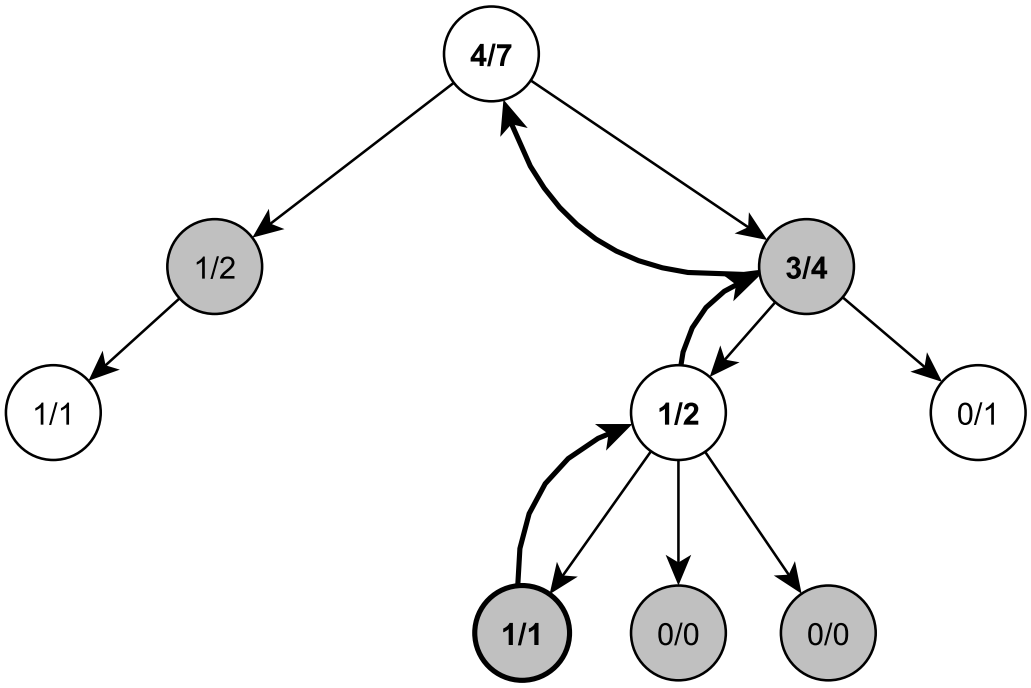


Obrázek 2.8: Simulace

Bílé vrcholy patří našemu hráči, šedé nepřátelskému, tučně zvýrazněnou hodnotu změnila simulace: ohodnotila vrchol hodnotou 1, tedy nepřátelský hráč vyhrál.

Zpětná propagace

Cílem zpětné propagace je aktualizovat strom hry podle výsledku simulace. Zpětná propagace nejprve aktualizuje list, ze kterého byla vedena simulace, jejím výsledkem, poté tuto hodnotu vypropaguje zpět až do kořene, zatímco aktualizuje také hodnoty všech vrcholů po cestě z listu do něj.



Obrázek 2.9: Zpětná propagace

Bílé vrcholy patří našemu hráči, šedé nepřátelskému. Tučně zvýrazněný vrchol je vrchol, ze kterého byla vedena simulace a je nyní zdrojem zpětné propagace. Tučný text je text upravený zpětnou propagací, šipky naznačují směr, kterým se nové hodnoty šíří.

Shrnutí algoritmu

Monte Carlo tree search by se dal shrnout následujícím pseudokódem. Ten opakuje 4 fáze Monte Carlo tree search, dokud není splněna ukončovací podmínka (tou může být například časový limit). Poté vrátí nejlepší tah. To je typicky hrana do vrcholu, který je potomek kořene a má největší počet navštívení, neboli počet her.

```
najdiNejlepšíTah(kořen) : tah
    opakuj
        // selekce
        vrchol := selekcePotomka(kořen);
        // expanze
        novýList := expanduj(vrchol);
        // simulace
        číslo := simuluj(novýList);
        // zpětná propagace
        propaguj(novýList, číslo);
    dokud není splněna ukončovací podmínka

    // vyber nejlepší tah
```

```
vrať vyberNejlepšíTah(kořen);
```

Algoritmus 2.4: Monte Carlo tree search

2.3.3 Volba algoritmu

Pro naši práci byl zvolen algoritmus Monte Carlo tree search. Je vhodnější, protože dává prioritu v prohledávání lepším pokračováním. U her s velkou složitostí, jako je Warlight, toto představuje obrovskou výhodu. Vyšší paměťovou složitost MCTS se budeme snažit vyřešit chytrou heuristikou, která umožní procházení pouze některých tahů.

3. Umělá inteligence

V této kapitole ukážeme náš přístup k tvorbě umělé inteligence a popíšeme její implementaci. Následně otestujeme její schopnosti v experimentech a výsledky experimentů zanalyzujeme.

3.1 Monte Carlo tree search ve Warlightu

V této sekci nejprve zanalyzujeme problémy algoritmu Monte Carlo tree search spojené s jeho použitím ve hře Warlight. Poté popíšeme úpravy jeho základní varianty z 2.3.2 tak, aby lépe fungoval ve hře Warlight.

3.1.1 Analýza

Cílem této sekce je prozkoumat problémy algoritmu Monte Carlo tree search ve hře Warlight.

Simulace

Z analýzy 2 víme, že hra Warlight má obrovský větvící faktor. Ukazuje se však také, že velká většina z možných tahů je velmi špatná.

Představme si, že jsme žlutý hráč na obrázku.

Obrázek 3.1: Analýza simulace

Obsadili jsme tři ze čtyř regionů super reginou Jižní Amerika. Zbývá nám dobýt jeden na dosažení bonusu. Nepřítele pro jednoduchost nebereme v úvahu. Zřejmě správným postupem je pokusit se o dobytí posledního regionu Jižní Ameriky. Počet dalších akcí, které můžeme provést, je ale z analýzy 2 nesmírně velký,

a tedy šance, že při odehrávání zvolíme právě toto správné pokračování, je velmi malá.

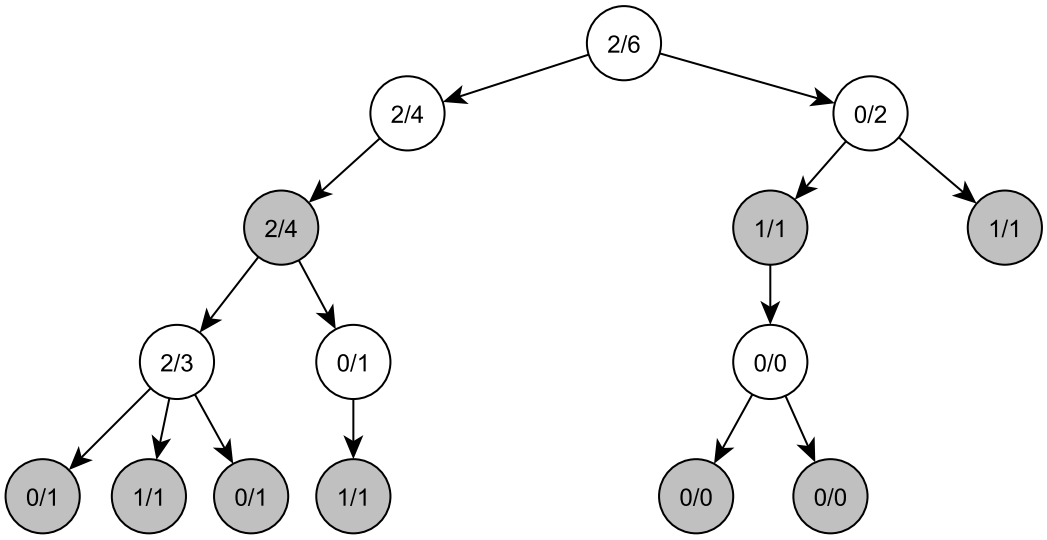
Při náhodném odehrávání se také počet kol hry velmi zvyšuje. Výpočet nového kola navíc vyžaduje poměrně hodně času, protože je potřeba vypočítat všechny změny způsobeny boji armád.

Jeví se, že náhodné odehrávání tahů není dobrý nápad a výhodnější je použít těžkou simulaci (heavy playout) ze sekce 2.3.2.

3.1.2 Strom hry

U každého vrcholu upravíme definici jeho vlastníka. Důvodem k tomu je snazší implementace algoritmu. Vlastníkem bude nově hráč, který odehrál tah vedoucí do tohoto vrcholu. Vlastníkem kořene a jeho dětí bude náš hráč. Od následujících úrovní stromu se bude vlastnictví vždy střídat začínaje od nepřítele.

Stav hry stačí mít uložený v kořeni a ve vrcholech vlastněných nepřítelem. Důvodem k tomu je fakt, že ve Warlightu nejprve všichni hráči odehrají své tahy a až poté dojde k výpočtu kola. Stačí si tedy pamatovat stav pouze ve vrcholu, ve kterém už všichni hráči odehráli své tahy. Protože (ignorujeme-li kořen) při pohledu od vrchu stromu je vždy nejprve náš vrchol a potom vrchol nepřítele (z čehož plyne nejprve náš tah a potom tah nepřítele), stačí si pamatovat stav (až na kořen) pouze ve vrcholech nepřítele.

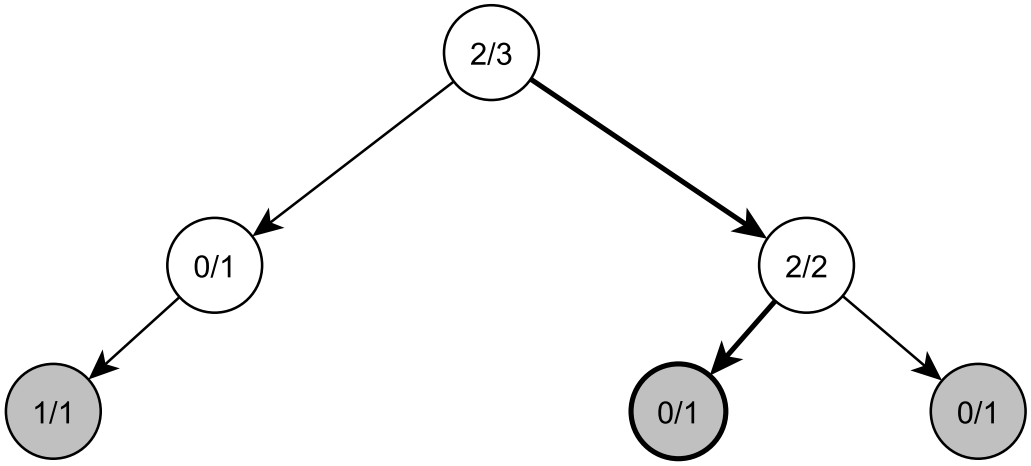


Obrázek 3.2: Upravený strom hry

Bílé vrcholy jsou vrcholy našeho hráče, šedé vrcholy jsou hráče nepřátelského. Z toho plyne, že hrany jdoucí do bílých vrcholů jsou tahy našeho hráče a hrany jdoucí do šedých vrcholů tahy hráče nepřátelského. Stav hry mají v sobě uložené pouze šedé vrcholy a kořen. Čísla vepsaná ve vrcholech m/n představují *počet Výher/počet Her* daného vrcholu.

3.1.3 Selekce

Princip fungování selekce je stejný jako u obecného MCTS algoritmu popsaného v sekci 2.3.2. Liší se ve tvaru stromu, na kterém je selekce prováděna. Navíc je pro selekci využita modifikovaná varianta *UCT*, která ve vzorci místo počtu her rodiče vrcholu N_i používá počet her v kořeni N.

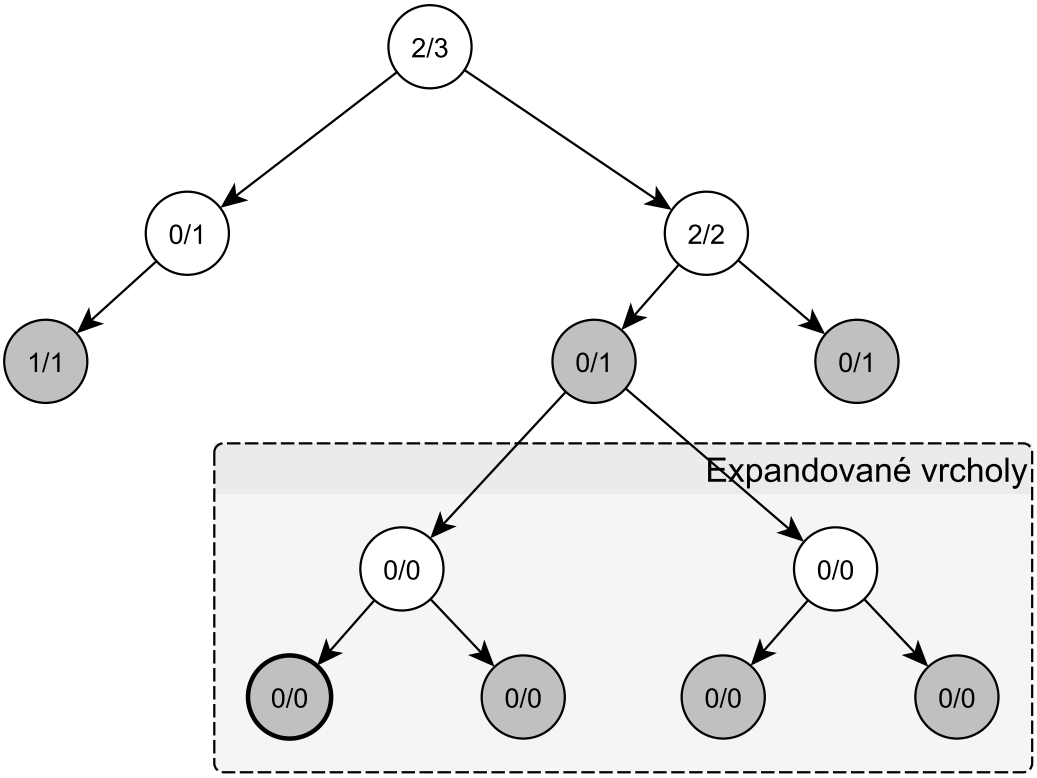


Obrázek 3.3: Selekce

Tučně zvýrazněné šipky ukazují hrany, po kterých selekce sestupuje z kořene do listu. Tučně zvýrazněný vrchol je list, který byl selekcí zvolen.

3.1.4 Expanze

Z tvaru stromu hry víme, že nový stav mají v sobě uloženy pouze nepřátelské vrcholy a kořen. Proto nemá smysl v expanzi přidávat potomky po jedné hladině, neboť následnou simulací bychom prováděli z vrcholu, který nemá svůj vlastní stav, což nedává smysl. Tomu je potřeba základní algoritmus expanze přizpůsobit. Metoda `přidejPotomky` z algoritmu 2.2 nejprve listu přidá naše vrcholy jako potomky, poté těmto novým potomkům přidá ještě nepřátelské vrcholy. Tím nám vzniknou nové dvě úrovně stromu, kde první úroveň bude náležet našemu hráči a druhá hráči nepřátelskému. Nové kolo bude vypočteno pouze v nových vrcholech druhé úrovně.



Obrázek 3.4: Expanze

Bílé vrcholy představují vrcholy našeho hráče, šedé nepřátelského hráče. Šedé vrcholy expandovaného stromu vždy drží změněný stav hry po odehrání všech tahů. Vrchol označený tučně zvýrazněným rámečkem je vrácen expanzí.

Dosud jsme předpokládali, že při expanzi dokážeme vygenerovat tahy. Jak bylo namítnuto v analýze, tento úkol není tak úplně snadný. Z analýzy víme, že tahy není dobré generovat zcela náhodně. Je potřeba zavést metodu na generování dobrých tahů. Tento problém řeší generátory akcí.

Generátory akcí

Generátor akcí je komponenta, jejímž účelem je nalézt (pokud možno malou) množinu smysluplných tahů pro daný stav hry a hráče. Generátory akcí dělíme na generátory akcí na začátku hry a generátory akcí pro ostatní fáze hry.

Generátory akcí na začátku hry Generátory akcí na začátku hry mají za úkol vygenerovat množinu smysluplných tahů skládajících se z akcí obsazujících regiony. Náš akční generátor pro začátek hry bere regiony od nejlepšího, dokud nenarazí na omezení v počtu regionů, které může na začátku hry obsadit.

Metoda `vygenerujTahy` níže implementuje algoritmus generování akcí na začátku hry. Používá k tomu další metodu `zvolNejlepšíTahy`. Ta rekurzivně volí nejlepší regiony. Jakmile zvolí jejich maximální počet, zvolené regiony přiřadí do výsledných tahů.

```
zvolNejlepšíTahy(stavHry, tahy, zvolenéRegiony)
    pokud byl zvolen max počet regionů
        tahy.přidej(zvolenéRegiony)
        vrať

    // setřiď sestupně regiony podle hodnoty
    regiony := setřiďRegiony(stavHry.regiony)

    pro každý region r z regiony
        // pokud r byl zvolen, nechceme ho volit vícekrát
        pokud r je ve zvolenéRegiony
            pokračuj

        // obsaď region
        r.vlastník = hráčNaTahu
        zvolenéRegiony.přidej(r)

        // rekurzivně zavolej tu stejnou funkci
        zvolNejlepšíTahy(stavHry, tahy, zvolenéRegiony)

        // vrať do původního stavu
        // před zavoláním algoritmu
        zvolenéRegiony.odeber(r)
        r.vlastník = původníVlastník

vygenerujTahy(stavHry) : tahy
    tahy := {}

    zvolNejlepšíRegiony(stavHry, tahy)

    vrať tahy
```

Algoritmus 3.1: Generování akcí na začátku hry

Generátory akcí pro pozdější fáze hry Generátory akcí pro pozdější fáze hry generují smysluplné tahy skládající se z deploy a attack akcí. Náš akční generátor pro pozdější fáze hry nejprve vygeneruje deploy sekvence, a pro každou z nich poté vygeneruje attack sekvence. Výsledkem je kartézský součin deploy a attack sekvencí, ve kterém jsou posléze odstraněny duplikáty. Algoritmus tak místo generování náhodných permutací akcí v tahu používá několik potenciálně dobrých pokračování, která vzájemně kombinuje a znatelně tak zmenšuje stavový prostor. Nevýhodou tohoto přístupu je oddělení generování deploy a attack akcí, výhodou je však jednodušší implementace.

Následující kód popisuje výše zmíněný postup. Algoritmus nejprve vygeneruje potenciálně dobré deploy sekvence. Poté ke každé deploy sekvenci přiřadí možné attack sekvence. Jejich kombinace jsou pak přidány do množiny výsledných tahů. Poté jsou odstraněny duplikáty, protože použití s nimi by vedlo k zbytečně vyššímu větvení stromu hry. Nakonec jsou tahy zpřeházeny pro zachování náhodnosti algoritmu.

```
vygenerujTahy(stavHry) : tahy
    tahy := {}

    // vygeneruje možnosti, jak udělat deploy
    deploySekvence := vygenerujDeploy(stavHry)

    pro každou deploy z deploySekvence
        // přehrej deploy sekvenci akcí
        aktualizovanýStav
            := přehrejDeploy(stavHry, deploy)

        // vygeneruj možnosti jak zaútočit pro daný deploy
        útoky := vygenerujMožnostiÚtoku(aktualizovnýStav)

        // přidej všechny kombinace deploy a útoků do tahů
        pro každý útok z útoky
            tahy.přidej(deploy, útok)

    odstraňDuplikáty(tahy)
    náhodněZpermutuj(tahy)
    vrať tahy
```

Algoritmus 3.2: Generování akcí pro pozdější fáze hry

Zbývá popsat, jak ohodnotit kvalitu regionu (kvůli algoritmu 3.1 výše) a generování deploy a attack akcí. Generování deploy a attack akcí však vyžadují schopnost ohodnotit současnou pozici a její jednotlivé komponenty. Uděláme proto menší odbočku do ohodnocovacích funkcí.

Ohodnocovací funkce

Generátory akcí pro své fungování vyžadují schopnost ohodnotit stav. Cílem ohodnocovací funkce je získat co nejlepší představu o kvalitě pozice z pohledu daného hráče. Tato představa je získána určením číselné hodnoty vyjadřující tuto kvalitu. Na této hodnotě pro daného hráče by se měla podílet hodnota jeho regionů a množství jeho jednotek.

Algoritmus níže ohodnotí pozici každého hráče tak, že sečte hodnotu všech jeho regionů a armád.

```
ohodnoťPoziciHráče(hráč) : hodnota
    hodnoceníPoziceHráče := 0;
    pro každý region takový, že hráč ho vlastní
        hodnoceníPoziceHráče += ohodnoťRegion(region)
            + c * region.armáda;

    vrať hodnoceníPoziceHráče;
```

Algoritmus 3.3: Ohodnocení pozice hráče

, kde c je reálná konstanta.

Algoritmus 3.3 pro své fungování požaduje funkci `ohodnoťRegion`. Následující odstavce adresují právě tento problém.

Ohodnocení regionu Ohodnocení regionu musí být odlišné pro začátek hry, kde jsou vybírány počáteční regiony a zbylé části hry. Důvodem k tomu je fakt, že se tyto dvě části velmi liší.

Při začátku hry platí, že:

- *Není dobré brát regiony blízko k sobě* - jedna z možných strategií, která každého napadne je, vzít více regionů blízko sebe (ideálně od stejného super regionu) a soustředit se na obsazení jednoho konkrétního super regionu všemi jednotkami. Pokud by se soupeři podařilo zabránit nám v jeho získání a zároveň obsazovat super region na druhém konci mapy (což může, protože bránící hráč má menší ztráty (1.5.2)), zcela jistě by vyhrál. Pokud by se to soupeři nepodařilo, stále se může rozvinout v jiném místě a porazit nás. Získává tak druhou šanci, kterou my nikdy s touto strategií nezískáme.
- *Hodnota super regionu přidává na hodnotě regionu* - zřejmé. Náleží-li region k lepšímu super regionu, je tento region tím cennější. Ohodnocení super regionu je popsáno později v textu.

Následující algoritmus formuluje výše zmíněná pravidla do ohodnocovací funkce. Nejprve přiřadí hodnotu získanou ohodnocením super regionu. Následně pak přičte k této hodnotě minimum spočítané ze vzdálenosti našeho nejbližšího regionu a polovině maximální vzdálenosti dvou regionů na mapě (polovině, aby region nebyl příliš zvýhodněn, protože je na opačném konci mapy – polovina maximální vzdálenosti na mapě je dostatečné maximum). Tato polovina je v algoritmu použita jako dělení 2.

```
ohodnoťRegionPřiZačátku(region, náš Hráč) : číslo
    hodnota := 0;

    // přičti hodnotu super regionu regionu
    hodnota += a
        * hodnotaSuperRegionu(region.superRegion);

    pokud jsem již zvolil nějaký region
        našeRegiony := náš Hráč.regiony;
        // spočti minimální vzdálenost k
        // jakémukoliv našemu regionu
        minimálníVzdálenostKRegionům
            := minVzdálenost(našeRegiony, region);

        // přičti shora omezenou hodnotu
        // za blízkost k regionu
        // čím bližší, tím menší bonus
        hodnota += b
            * min(minimálníVzdálenostKNašemuRegionu,
            maximálníVzdálenostDvouRegionůNaMapě / 2);
```

```
vrať hodnota;
```

Algoritmus 3.4: Ohodnocení regionu na začátku hry

, kde a a b jsou reálné konstanty.

Po započetí hry je ohodnocovací funkce odlišná. Platí, že:

- *Hodnota super regionu přidává na hodnotě regionu* - zřejmé. Čím lepší super region, tím lepší region, který k němu náleží.
- *Výhodnější je vlastnit u super regionu co nejvíce regionů, nejlépe celý super region* - našim dílčím cílem je obsadit celý super region. To nám zajistí vyšší příjem jednotek za kolo, potažmo výhodu nad nepřítelem. Čím více regionů ze super regionu máme, tím motivovanější bychom měli být obsadit zbytek. Nejvýhodnější je pak samozřejmě mít celý super region kvůli jeho bonusu.
- *Region, který patří nějakému hráči, má větší hodnotu[6]* - pokud obsazujeme neutrální území místo abychom bojovali s nepřítelem, ztrácíme přitom jednotky zatímco náš nepřítel ne. Je potřeba upřednostnit boj s nepřítelem před bojem s neutrálním hráčem.

Algoritmus níže tato pravidla opět formuluje do vzorce. Zpočátku přiřadí do výsledku ohodnocení super regionu, poté přičte bonus za počet mnou obsazených regionů tohoto super regionu. Vlastní-li někdo tento super region celý, přičte bonus. Navíc, pokud region patří nepříteli, přičte bonus za všechny regiony super regionu vlastněné nepřítelem. Nakonec vynásobí konstantou počítanou hodnotu, pokud region není neobsazený.

```
ohodnoťRegion(region, náš Hráč) : číslo
    hodnota := 0;
    // připočti hodnotu super regionu
    hodnota += a
        * hodnotaSuperRegionu(region.superRegion);

    // přidej bonus za počet regionů pro našeho hráče
    hodnota += b
        * početRegionůSuperRegionu(náš Hráč,
            region.superRegion);

    // přičti bonus, pokud je obsazený celý super region
    pokud super region patří nějakému hráči celý
        hodnota += c
            * získejBonusZaSuperRegion(
                region.superRegion);

    // region patří nepříteli => připočti
    // bonus za to, že patří nepříteli
    pokud region.vlastník je soupeř
        hodnota += d
        * získejPočetRegionůSuperRegionu(
            nepřítel, region.superRegion);
```

```
// pokud patří nám nebo soupeři,
// vynásob hodnotu regionu konstantou
pokud region.vlastník jsme my nebo soupeř
    hodnota *= e;

vrať hodnota;
```

Algoritmus 3.5: Ohodnocení regionu po začátku hry

, kde a, b, c, d, e jsou reálné konstanty.

Pro ohodnocení regionu je však stále potřeba ohodnotit super region.

Ohodnocení super regionu Při ohodnocování super regionu bychom měli dbát na následující pravidla:

- *Je výhodné brát super region, který má málo sousedících regionů[4]* - takový region se po dobytí se bude lépe bránit.
- *Lepší je super region s více sousedními super regiony[4]* - můžeme narušovat bonusy nepřátelům nebo rychle dobývat.
- *Je lepší super region s vyšším bonusem jednotek* - zřejmé.
- *Je nevýhodné brát super region, který se skládá z mnoha regionů[4]* - takový super region je těžké dobýt a trvá to dlouho. Nejjistější strategií je získat rychle bonusy.

Tato pravidla platí jak na začátku, tak i v pozdějších fázích hry. Pro pozdější fáze hry je potřeba přidat pouze jedno pravidlo – pokud jsme obsadili celý region, získáme bonus do ohodnocení. Implementace funkce pro ohodnocení super regionu je, podobně jako u regionu, vážený součet.

Nyní jsme vyřešili ohodnocování pozic a můžeme se vrátit ke generování akcí.

Generátor deploy akcí

Generátor deploy akcí je komponenta, která má vytvářet sekvence smysluplných deploy akcí (takových, že má význam je dále prozkoumávat). Využívá tři přístupy: útočný, obranný a expanzivní. Ty spolu formují základní strategie, které bot ve hře Warlight potřebuje: potřebuje útočit na nepřátelského hráče, bránit se jeho útokům a expandovat na neobsazené regiony. Níže je jejich podrobnější popis.

Útočný Postaví všechny možné jednotky na náš region sousedící s nejcennějším nepřátelským regionem, kde cenou regionu je výsledek ohodnocovací funkce 3.1.4 tohoto regionu. Tím zajistí možnost ve fázi útoku zaútočit s novými jednotkami poblíž nepřátelského regionu a obsadit tak nepřátelské území.

```
vygenerujAgresivníDeployAkci(stavHry, nášHráč) : deployAkce
    nepřátelskýRegion
        := nejcennějšíSousedníNepřátelskýRegion(
            stavHry, nášHráč)
```

```
    náš Útočící Region
        := vyberNášRegionSNejvíceJednotkami(
            nepřátelský Region.sousedi, náš Hráč);

    deployAkce := {náš Útočící Region,
        nepřátelský Region, maximální PočetJednotek};

    vrať deployAkce;
```

Algoritmus 3.6: Generování agresivních deploy akcí

Obranný Postaví jednotky na nejcennější regiony takové, kterým hrozí dobytí nepřítelem. Smyslem je zabránit dobytí pro nás důležitých regionů. V tomto případě je cennost určena komplexnějším měřítkem. Regiony, které patří k námi vlastněnému super regionu, jsou považovány automaticky za nejcennější. To zajistí zcela prioritní ochranu bonusu ze super regionu. Dále je pak cennost určena ohodnocovací funkcí.

Poté je potřeba regiony rozdělit podle toho, zda jsou v ohrožení či ne. Jako region v ohrožení definujeme takový region, který v případě, že nepřítel postaví veškeré své jednotky na některý z jeho sousedů, a všemi jednotkami pak zaútočí, tak ho v očekávaném výsledku dobude. Pokud region v ohrožení je, postavíme na něj minimální množství jednotek tak, že potenciální útok od nepřítele odrazí.

```
vygenerujObrannéDeployAkce(stavHry, náš Hráč) : deployAkce
    // vyber pouze regiony sousedící s nepřítelem
    regiony := vyberRegionSousedícíSNepřítelem(
        stavHry.regiony, náš Hráč)

    // nejprve seřaď podle toho,
    // jestli region patří k našim regionům,
    // potom podle ohodnocení sestupně
    regiony = seřaďRegionyPodleStupněOhrožení(regiony)

    deployAkce := {}
    pro každý region z regiony
        // pokud nepřítel může dobýt region,
        // jestliže postaví všechny jednotky na některý
        // svůj sousedící region
        pokud region je v ohrožení
            // přidej minimální počet jednotek tak,
            // že tyto útoky ubrání
            deployAkce
                .přidejMinPočetJednotekNaUbránění();
        pokud nemáme žádné další jednotky na postavení
            vrať deployAkce

    vrať deployAkce
```

Algoritmus 3.7: Generování obranných deploy akcí

Expanzivní V určitých herních situacích se nevyplatí ani útočit ani bránit, nýbrž expandovat. Představme si situaci, kdy na jednom konci mapy máme náš region sousedící s nepřítelem a na opačném konci mapy jsme obsadili skoro celý kontinent a chybí nám už pouze k dobytí jeden neobsazený region. Naše umělá inteligence by měla v těchto situacích rozpoznat příležitost k obsazení toho regionu, tedy i k obsazení celého super regionu a získání bonusu. To řeší expanzivní přístup. Ten postaví jednotky na našem regionu sousedící s nejcennějším neobsazeným regionem, kde cennost je opět určena ohodnocovací funkcí 3.1.4.

```
vygenerujExpanzivníDeployAkci(stavHry, náš Hráč)
                                        : deployAkce
    neobsazenýRegion
        := najdiNejcennější SousedníNeobsazenýRegion(
            stavHry, náš Hráč)
    náš Útočící Region
        := vyberNáš RegionSNejvíceJednotkami(
            neobsazenýRegion.sousedi, náš Hráč);

    deployAkce := {náš Útočící Region, neobsazenýRegion,
        maximální PočetJednotek};

    vrať deployAkce;
```

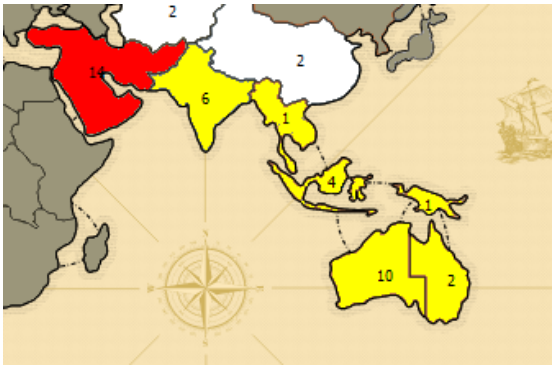
Algoritmus 3.8: Generování expanzivních deploy akcí

Generátor attack akcí

Generátor attack akcí je komponenta, která má vytvářet sekvence smysluplných attack akcí. Smysluplná attack akce je taková útočná akce, že má význam se jí zabývat.

Nejprve však definujeme několik společných procedur, které generátor attack akcí používá.

Přesun armády z vnitrozemí Uvažujme, že jsme žlutý hráč v situaci na obrázku 3.5. Zřejmě náš region sousedící s červeným regionem by potřeboval více jednotek. Tyto jednotky zbytečně čekají na regionech obklopených pouze námi vlastněnými regiony. Budeme je nazývat regiony *vnitrozemními*. Potřebujeme přesunovat armády z vnitrozemních regionů do regionů nevnitrozemních. Ty budeme nazývat regiony *krajními*.



Obrázek 3.5: Vnitrozemní regiony

Následující algoritmus implementuje tento postup. Pro každý námi vlastněný region zkontroluje, zdali je vnitrozemní. Pokud ano, pak najde cestu k nejbližšímu krajnímu regionu a vydá se po ní.

```
přesuňArmádyZVnitrozemí(stavHry, nášHráč)
    přesuny := {}
    pro každý náš region
        pokud všichni sousedi regionu jsou naše regiony
            // najdi nejbližší region, co není náš
            cizíRegion := najdiCizíRegion(region)

            // najdi cestu k němu
            cesta := najdiNejkratšíCestuMezi(
                region, cizíRegion)

            // najdi první region na této cestě
            prvníRegionNaCestě := cesta[1]

            přesun := pošliJednotky(
                region, prvníRegionNaCestě)

            přesuny.přidej(přesun)
```

Algoritmus 3.9: Přesun armády z vnitrozemí

Agresivní útok na nepřítele Agresivním útokem na nepřítele nazveme sekvenci attack akcí takových, že každá má za cíl dobýt nepřátelský region nebo alespoň mít při jeho získávání menší ztráty než nepřátelská armáda.

Algoritmus se pro každý náš region podívá na sousední nepřátelské regiony a vyhodnotí, zdali budeme mít při útoku z nich v očekávané hodnotě menší ztráty. Pokud ano, zaútočíme s minimálním počtem jednotek tak, aby dobyl nepřátelský region.

```
zaútočAgresivně(stavHry, nášHráč) : útok
    pro každý náš region r
        seřaď sousedy regionu r podle jejich hodnoty
        pro každého souseda regionu r n
            pokud n je nepřítelovo a naše armáda
```

```
            na regionu r bude mít při útoku
            menší ztráty
                zaútoč z r na n takovým počtem
                jednotek, abychom optimálně
                dobyli region n
```

Algoritmus 3.10: Agresivní útok

Opatrný útok na nepřítele Cílem opatrného útoku nepřítele je zaútočit na nepřátelské regiony tak, že i pokud by nepřítel postavil na kterýkoliv jeho region všechny jeho dostupné jednotky, i přesto tento boj vyhrajeme a daný region obsadíme.

```
zaútočOpatrně(stavHry, náš Hráč) : útok
    pro každý náš region r
        seřaď sousedy regionu r podle jejich hodnoty
        pro každého souseda regionu r n
            pokud n je nepřítelovo
            a při útoku z r na n budeme mít očekávaně
            menší ztráty, i když na region n
            postaví všechny dostupné jednotky
                zaútoč z r na n tak, abychom v
                očekávaném výsledku (i při postavení
                všech jednotek na n) vyhráli
```

Algoritmus 3.11: Opatrný útok

Opatrný expanzivní útok Opatrný expanzivní útok útočí na neobsazený sousední region v takovém případě, že region ze kterého útočíme nemá jako souseda nepřátelský region. To umožní obsazovat sousední regiony takovým způsobem, že nehrozí překvapivý útok od nepřítele.

```
zaútočExpanzivně(stavHry) : útok
    pro každý náš region r
        pokud region r nemá jako souseda
        nepřátelský region
            seřaď sousedy regionu r dle jejich hodnoty
            pro každého souseda n
                pokud při útoku z r na n
                budeme mít menší ztráty
                    zaútoč s co nejméně jednotkami
                    tak, abychom dobyli region n
```

Algoritmus 3.12: Opatrný expanzivní útok

Při generování attack akcí používáme tři přístupy: útočný, útočný s vyčkáním, obranný.

Útočný Z každého regionu se vždy podíváme na sousední nepřátelské nebo neobsazené regiony, a potom na ně začneme posílat útoky (algoritmus 3.10). Poté přesuneme jednotky z vnitrozemních regionů k regionům krajním (algoritmus 3.9).

```
generátorÚtočnéhoPřístupu(stavHry, hráč)
    zaútočAgresivně(stavHry, hráč)

    přesuňJednotkyNaOkraj(stavHry, hráč)
```

Algoritmus 3.13: Útočný přístup ke generování attack akcí

Útočný s vyčkáním Na začátku přesuneme jednotky z vnitrozemí (algoritmus 3.9). Poté zaútočíme na sousední neobsazené regiony takové, že nesousedí s nepřátelským regionem (algoritmus 3.12). Nakonec zaútočíme na nepřítele stejně jako v útočném přístupu (algoritmus 3.10).

To, že nejprve přesuneme jednotky a zaútočíme na neobsazené regiony, nám zajistí výhodu obránce v případě, že by nepřítel zaútočil jako první[4].

```
generátorÚtočnéhoPřístupuSVyčkáním(stavHry, hráč)
    přesuňJednotkyNaOkraj(stavHry, hráč)

    zaútočAgresivně(stavHry, hráč)
```

Algoritmus 3.14: Útočný přístup s vyčkáním ke generování attack akcí

Obranný Nejprve přesuneme jednotky z vnitrozemí (algoritmus 3.9), potom zaútočíme na sousední neobsazené regiony (algoritmus 3.12). Poté začneme posílat útoky opatrně: provádíme je tak, že na region zaútočíme jen s armádou, která porazí tu nepřátelskou i v případě, že by na něj nepřátelský hráč postavil v deploy fázi všechny dostupné jednotky (algoritmus 3.11).

```
generátorÚtočnéhoPřístupuSVyčkáním(stavHry, hráč)
    přesuňJednotkyNaOkraj(stavHry, hráč)

    zaútočExpanzivně(stavHry, hráč)

    zaútočOpatrně(stavHry, hráč)
```

Algoritmus 3.15: Obranný přístup ke generování attack akcí

Použitím výše uvedených algoritmů se nám větvící faktor značně zmenšil. Z původního odhadovaného větvícího faktoru $10^{13 \cdot 2} = 10^{26}$ (ze sekce 2.1) dojde ke zmenšení na $3 \cdot 3 = 9$ (3 přístupy ke generování deploy akcí, 3 přístupy ke generování attack akcí). Často je ale větvení menší, protože některé z tahů nemusí být unikátní a algoritmus 3.2 odstraňuje duplikáty.

Vyhodnocování nového kola

Při expanzi je ve druhé úrovni z přidaných vrcholů uložena nová pozice. Ta je určena vygenerovanými tahy hráčů, které tvoří kolo a odehráním tohoto kola. Odehrávání kol je ale do značné míry náhodné, protože jednotky mají procentuální šanci na zabití nepřátelské jednotky. Můžeme se dostat tedy do mnoha možných stavů hry. Abychom nezvyšovali větvící faktor, potřebujeme z nich vybrat.

Vyhodnotíme tedy kolo n-krát. Nové stavy hry pak seřadíme podle hodnocení z pohledu našeho hráče. Z těchto výsledků zvolíme jeden nejvhodnější stav, který určíme jako stav reprezentující výsledek odehraného kola.

```
výpočetStavuNovéhoKola(nášTah, nepřTah, nášHráč) : stav
    výsledky = {}

    pro i := 0 do n opakuj
        novýStav := vyhodnoťNovéKolo(nášTah, nepřTah)
        hodnotaNovéhoStavu
            := ohodnocenízPohledu(nášHráč, novýStav)

        výsledky.přidej({novýStav, hodnotaNovéhoStavu})

    // seřaď podle hodnoty nového stavu
    výsledky := výsledky.seřaďPodle(hodnotaNovéhoStavu)

    // vyber vhodný stav
    vrať vyberVhodnýVýsledek(výsledky)
```

Algoritmus 3.16: Výpočet stavu nového kola

Máme tři přístupy volby vhodného výsledku odehrání kola: optimistický, realistický a pesimistický.

1. *Optimistický* - vezme stav nejvýhodnější pro nás,
2. *Realistický* - vezme medián,
3. *Pesimistický* - vezme stav pro nás nejméně výhodný.

Optimistický je na první pohled špatný. Je zřejmé, že se nevyplatí zkoumat větve, kde jsme při výpočtu měli štěstí. Pro většinu situací se vyplatí realistický přístup.

V některých situacích se ale vyplatí pesimistický přístup. Pokud by například hrozilo, že přijdeme o bonus za super region, realistický přístup by mohl vzít stav, kde bychom o něj nepřišli. Následný výpočet algoritmu by se touto možností už nezabýval a MCTS by mohlo zvolit pokračování, ve kterém bychom tento super region ztratili.

Níže uvedený algoritmus implementuje metodu `vyberVhodnýVýsledek` použitou v algoritmu 3.16.

```
vyberVhodnýVýsledek(výsledky) : stav
    pesimistický := zvolNejhorší(výsledky)
```

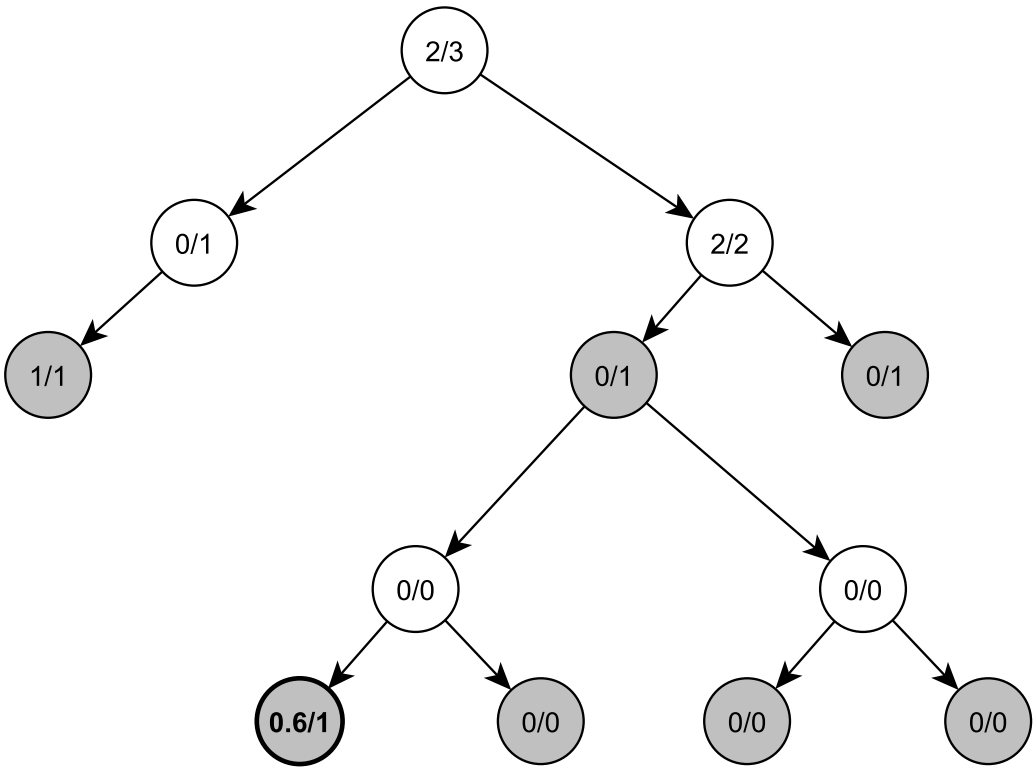
```
realistický := zvolMedián(výsledky)

pokud jsem v pesimistickém přišel o super region
    vrať pesimistický

vrať realistický
```

Algoritmus 3.17: Volba nového stavu

3.1.5 Simulace

Z analýzy (3.1.1) je zřejmé, že je výhodnější použít těžkou simulaci místo lehké, protože lehká by nemusela vůbec skončit, a pokud by skončila, tak by špatně odhadovala pozici, ze které byla vedena. Každý vygenerovaný tah při těžké simulace je však časově náročnější.

Řešením je odsimulovat předem určený počet kol a ohodnotit výslednou pozici. Místo původního ohodnocení výsledku simulace $0, 1$ ohodnotíme výsledek intervalem $[0, 1]$, kde v našem vrcholu 1 bude výhra našeho hráče a 0 výhra soupeře (pro ohodnocení na nepřátelském vrcholu je ohodnocení opačně). To zajistí přesnější odhad kvality pozice.



Obrázek 3.6: Simulace

Bílé vrcholy patří našemu hráči, šedé nepřátelskému, tučně zvýrazněnou hodnotu změnila simulace – ohodnotila vrchol hodnotou 0.6, tedy lehce výhodněji pro nepřátelského hráče.

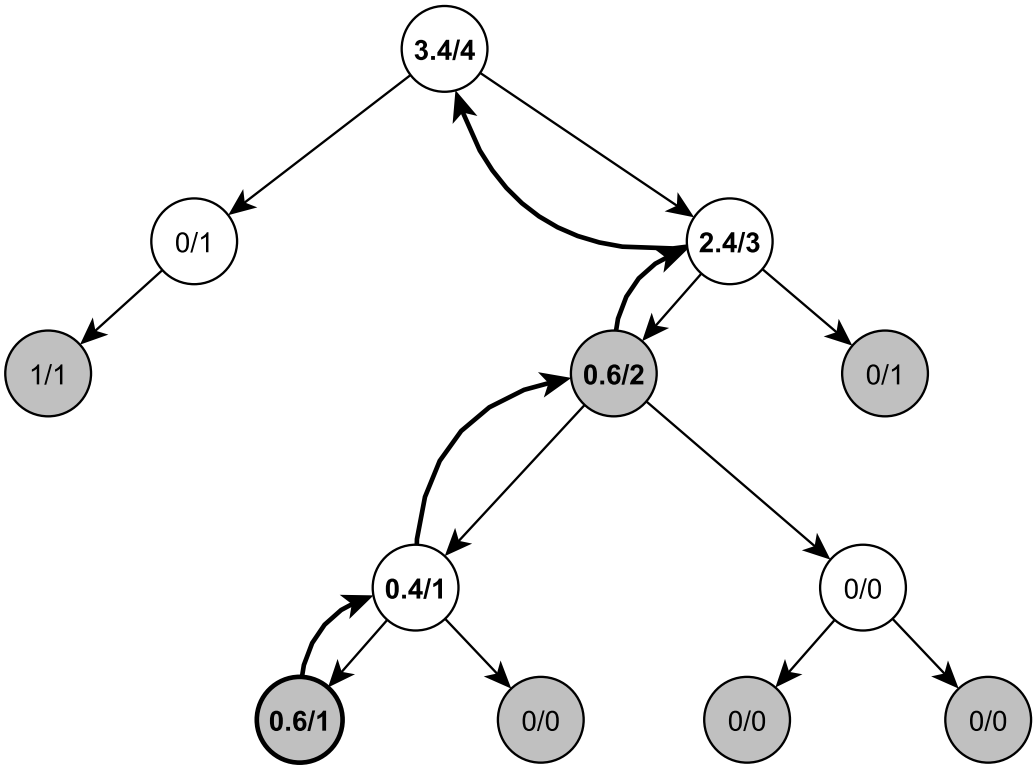
Simulace tedy odehraje n kol použitím generátoru akcí 3.1.4. Poté ohodnotí pozici z pohledu obou hráčů 3.1.4. Pozici z pohledu nepřátelského hráče pak normalizuje do intervalu $[0, 1]$ vydělením ohodnocení nepřátelského hráče součtem ohodnocení nepřátelského hráče a našeho hráče.

```
Simuluj(list, náš Hráč) : číslo
    stav := list.stavHry

    // iteruj přes počet kol simulace
    pro i = 0 do n
        pokud je konec hry a vyhráli jsme my
            vrať 0
        pokud je konec hry a vyhrál nepřítel
            vrať 1

        náš Tah := vygenerujTahy(stav, náš Hráč)[0]
        nepřátelský Tah
            := vygenerujTahy(stav, nepřátelský Hráč)[0]

        stav := výpočetStavuNovéhoKola(
            náš Tah, nepřátelský Tah)

    ohodnocení Nepř := ohodnoťStavHry(
        stav, nepřátelský Hráč)
    ohodnocení Naše := ohodnoťStavHry(stav, náš Hráč)

    // normalizuj ohodnocení
    vrať ohodnocení Nepř / (ohodnocení Naše + ohodnocení Nepř)
```

Algoritmus 3.18: Simulace

3.1.6 Zpětná propagace

Fáze propagace se liší především ve schopnosti práce s reálnými čísly reprezentujícími počet výher.



Obrázek 3.7: Zpětná propagace

Bílé vrcholy patří našemu hráči, šedé nepřátelskému. Tučně zvýrazněný vrchol je vrchol, ze kterého byla vedena simulace, a je nyní zdrojem zpětné propagace. Tučně zvýrazněné číslo je číslo upravené zpětnou propagací, šipky naznačují směr, kterým se nové hodnoty šíří.

Algoritmus níže implementuje zpětnou propagaci. Jako parametr přijme list, ze kterého byla vedena simulace a její ohodnocení. Poté na cestě z tohoto listu do kořene upravuje všechny vrcholy (včetně těchto dvou) tak, že upraví jejich počet výher a počet her. Pro vrcholy patřící nepříteli je k počtu výher přičtena přímo hodnota ze simulace. Pro naše vrcholy je k ní přičten doplněk do 1. Tento postup je možný díky vzorci z algoritmu 3.18, který používá pro výpočet výsledku simulace `ohodnoceníNepř / (ohodnoceníNaše + ohodnoceníNepř)`, kde `ohodnoceníNepř` je ohodnocení pozice nepřítele na konci simulace a `ohodnoceníNaše` je ohodnocení naší pozice na konci simulace.

```
zpětnáPropagace(list, ohodnoceníListu)
    vrchol := list

    dokud vrchol není null
        pokud vrchol patří nepříteli
            vrchol.početVýher += ohodnoceníListu
        jinak
            // vzorec pro výpočet ohodnocení listu
            // nám zajišťuje, že pro pozici platí
```

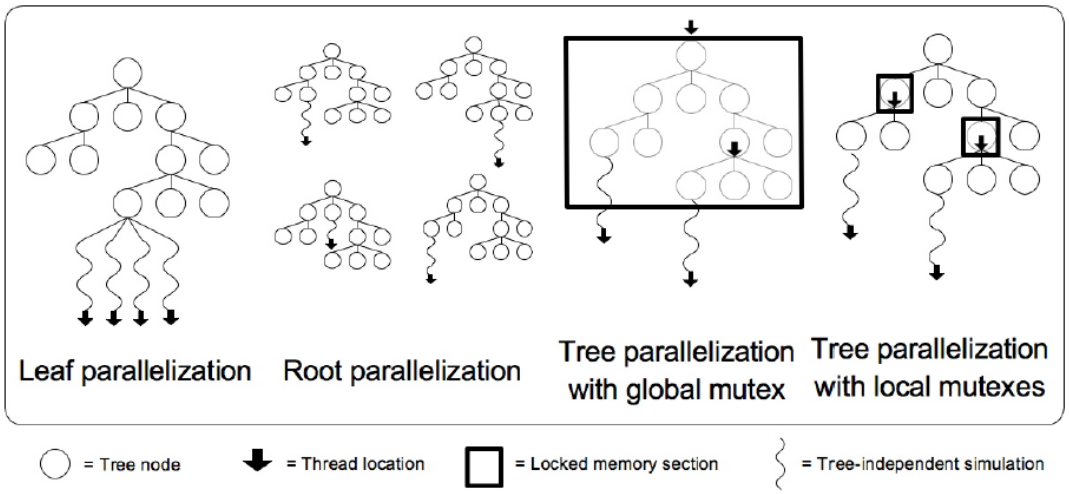
```
        // našeOhodnocení = 1 - nepřOhodnocení
        vrchol.početVýher += 1 - ohodnoceníListu

    vrchol.početHer++

    vrchol := vrchol.rodič
```

Algoritmus 3.19: Zpětná propagace

3.1.7 Paralelní MCTS

I přes znatelné zmenšení větvícího faktoru je stavový prostor hry stále velký. Pro vyšší výkon by bylo vhodné paralelizovat výpočet při hledání tahu. V této sekci prozkoumáme možnosti paralelizace algoritmu MCTS a zvolíme metodu přístupu k paralelizaci.

Chaslot a kol. [3] popisují čtyři druhy paralelizace: listovou, kořenovou, stromovou s globálním zámkem a stromovou s lokálními zámky.



Obrázek 3.8: Druhy paralelizace MCTS

Listová paralelizace Z expandovaného vrcholu je vedeno místo jedné simulace simulací několik. Tento postup je velmi jednoduchý na implementaci, avšak má pro nás jeden důležitý problém: rozehrajeme-li například 8 simulací, kde 7 skončí prohrou, pak čas výpočtu strávený na 8. z nich byl ztracený, neboť po 7 prohrách jsme mohli usoudit, že to nemá cenu dále zkoušet.

Kořenová paralelizace Místo stavění jednoho stromu hry jich stavíme rovnou několik. Tyto stromy jsou na sobě zcela nezávislé. Když vyprší čas a algoritmus má vrátit nejlepší nalezený tah, děti kořenů ve všech stromech jsou sjednoceny: jsou-li dva vrcholy stejné, sečte se jejich počet výher a počet her a slijí se do jednoho, a v těchto sjednocených vrcholech se vezme vrchol s nejvyšším počtem návštěv algoritmu. Tato paralelizace je opět velmi jednoduchá.

Stromová paralelizace s globálním zámkem Na strom je umístěn zámek, který zakáže přístup ke stromu více než jednomu vláknu. Simulace můžou probíhat paralelně, ostatní fáze MCTS však ne.

Stromová paralelizace s lokálními zámky Stromová paralelizace s lokálními zámky umožňuje přístup ke stromu více vláknům. Kdykoliv vlákno přistoupí k vrcholu stromu, zamkne si ho, jakmile ho opustí, odemkne ho. Budou-li použity jako zámky spinlocky, protože zamykání bude pouze na krátkou chvíli, tento přístup by měl být efektivní. Častokrát se však bude stávat, že vlákna budou procházet strom zcela stejným způsobem. Pokud vlákna začnou v selekci paralelně sestupovat od kořene až k listu, je pravděpodobné, že budou sestupovat po stejné cestě. Po odsimulování by vzestup byl také po stejné cestě. Tím by docházelo k častému čekání vláken na odemknutí vrcholů.

Aby se tomuto vyhnuli, Chaslot a kol. [3] využili koncept *virtuální prohry*. Kdykoliv vlákno přistoupí k vrcholu, přiřadí mu virtuální prohru, čímž zmenší jeho hodnocení. Následující vlákno tak nemusí jít nutně po té stejné cestě. Při zpětné propagaci je tato virtuální prohra zrušena.

Tyto metody paralelizace byly v práci Chaslot a kol. [3] otestovány na hře Go. Výsledky těchto metod jsou pro naši práci relevantní díky podobnosti těchto her ve velikosti stavového prostoru.

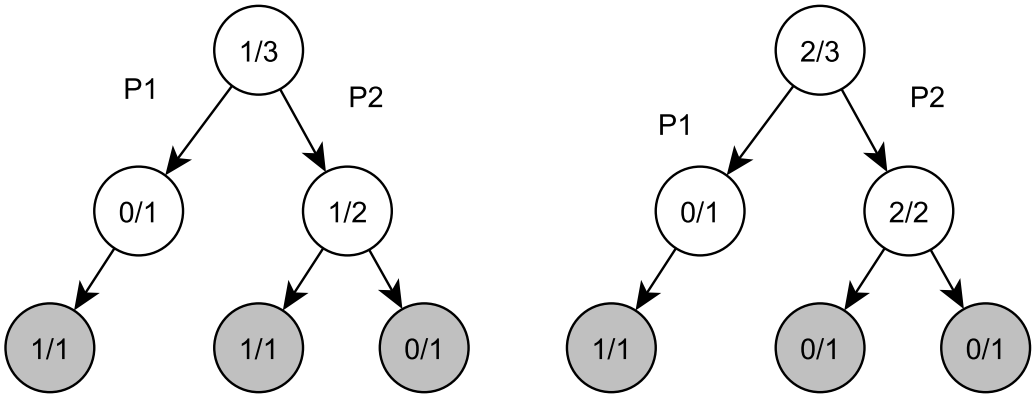
Počet vláken	**Počet výher**	**Počet her**
Listová paralelizace:		
1	26.7%	2000
2	26.8%	2000
4	32.0%	1000
16	36.5%	500
Kořenová paralelizace:		
1	26.7%	2000
2	38.0%	2000
4	46.8%	2000
16	56.5%	2000
Stromová paralelizace s globálním zámkem:		
1	26.7%	2000
2	31.3%	2000
4	37.9%	2000
16	36.5%	500
Stromová paralelizace s lokálními zámky:		
1	26.7%	2000
2	33.8%	2000
4	40.5%	2000
16	49.9%	2000

Tabulka 3.1: Výsledky testování jednotlivých přístupů k paralelizaci na hře Go[3].

Listová paralelizace se ukázala být neefektivní. Stromová paralelizace s lokálními zámky je spíše obtížnější na implementaci a ukazuje podobné výsledky

jako kořenová paralelizace. Proto jsme se v naší práci rozhodli pro kořenovou paralelizaci.

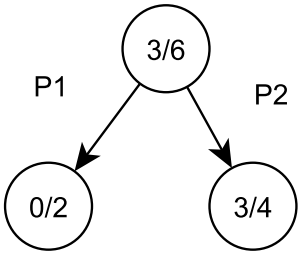
Využijeme ji tedy tak, že budeme stavět paralelně n stromů v závislosti na počtu vláken a výpočet na nich budeme provádět paralelně.



Obrázek 3.9: Stromová paralelizace ve Warlightu

Bílé vrcholy patří našemu hráči, šedé nepřátelskému, P1 a P2 představují tahy.

Po skončení výpočtu zjistíme, který tah se ukázal být pro nás nejlepší, neboli který měl nejvyšší počet her ve vrcholu, do kterého vede. To zjistíme sjednocením prvních dvou úrovní stromů podle akcí a sečtením počtu výher a her těchto vrcholů.



Obrázek 3.10: Spojené stromy

První dvě sjednocené úrovně stromů z obrázku 3.9.

3.2 Implementace umělé inteligence

Tato sekce se věnuje popisu naší implementace MCTS bota. Provedeme analýzu, kde určíme, jaké nároky budou na návrh umělé inteligence kladeny. Na jejich základě pak popíšeme naši implementaci.

3.2.1 Analýza

Cílem analýzy je zformulovat požadavky na implementaci umělé inteligence ve hře Warlight. Nejprve popíšeme požadavky na reprezentaci stavu hry a akcí,

poté na objektový návrh.

Reprezentace stavu hry a akcí

Stav hry potřebujeme během výpočtu poměrně často kopírovat, například při generování akcí, kde kombinujeme deploy a attack akce (sekce 3.1.4). Nejprve na základě určitého stavu potřebujeme vygenerovat deploy akce, pak na základě pozměněného stavu (stav po aplikované deploy akci) vygenerovat attack akce. Jiný akční generátor by však snadno mohl kopírovat stav ještě častěji. Tato operace by tedy měla být rychlá.

Algoritmus Monte Carlo tree search navíc může spotřebovávat poměrně hodně paměti. Staví strom hry, kde v každém jeho vrcholu si potřebuje zapamatovat akci, která do něj vede, v polovině z nich pak navíc stav celé mapy (sekce 3.1.1). Stavy hry a tahy (neboli akce) by tedy měly být reprezentovány tak, aby nezabíraly příliš mnoho paměti. Toto bude mít pozitivní efekt i na rychlost kopírování, protože malé struktury se rychleji kopírují.

Objektový návrh

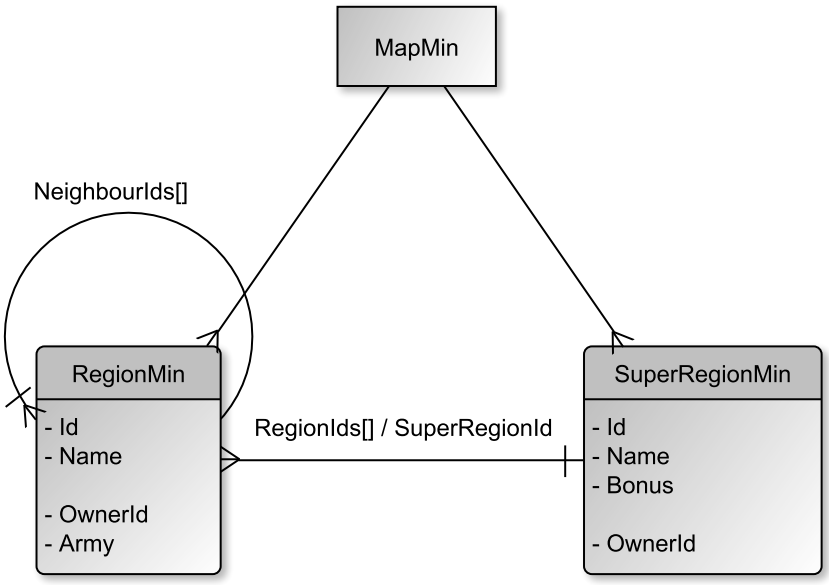
Od objektového návrhu bota očekáváme, že bude snadné přidat další implementaci bota.

Dále ohodnocovací funkce ze sekce 3.1.4 jsou implementovány pomocí váženého průměru použitím různých koeficientů. Proto by měla být umožněna jednoduchá změna těchto konfigurací ohodnocovacích funkcí.

Nakonec je třeba oddělit generování akcí od samotného bota, aby stejný generátor akcí šel použít pro několik typů botů. Navíc by generátor akcí pro samotnou instanci bota měl být snadno nahraditelný jiným.

3.2.2 Reprezentace stavu hry

Z analýzy 3.2.1 víme, že stav hry by měl být paměťově efektivní a rychlý na kopírování. V naší práci je reprezentován strukturou `MapMin`. Ta představuje minifikovanou verzi mapy. Obsahuje kompletní informaci o stavu hry.



Obrázek 3.11: Reprezentace stavu hry

Region máme zareprezentovaný objektem `RegionMin`. Aby bylo zajištěno efektivní kopírování, jeho položky jsou rozděleny na statické a dynamické. Statické jsou takové, které se v průběhu hry nemění – například Id regionu, jméno regionu nebo Id super regionu. Dynamické jsou takové, které se v průběhu hry naopak mění – například tedy velikost armády, Id hráče, který vlastní region. Statické položky není potřeba kopírovat, proto jsou uchovávány ve speciální třídě, na kterou se `RegionMin` odkazuje referencí. Díky tomu při běžném kopírování statické položky nejsou kopírovány, je kopírována pouze reference na ně.

Super region je reprezentovaný objektem `SuperRegionMin`. Ten, podobně jako region, dělí položky na statické a dynamické a využívá tak stejného principu pro efektivnější kopírování.

Tyto minifikované objekty nikdy neobsahují odkazy na jiný objekt, který se může v průběhu výpočtu změnit, například `RegionMin` neobsahuje odkaz na `SuperRegionMin`, nýbrž má pouze jeho Id. Důvodem k tomu je, že při mělkém kopírování (shallow copy) mapy by se nám rozbily reference. Kvůli tomu, že neukládáme reference se nám však některé operace zkomplikovaly. Nedokážeme v konstantním čase získat objekt `SuperRegionMin` na základě jeho Id. Namísto toho musíme projít všechny super regiony a najít ten správný. Rychlost této operace je však důležitá, protože se provádí velmi často. Proto zavedeme pravidlo – seznam regionů a super regionů bude `MapMin` pole indexované Id regionu a super regionu.

Velikost stavu Abychom určili velikost stavu, potřebujeme sečíst velikost všech jeho dílčích komponent. Víme, že stav se skládá ze seznamu objektů `RegionMin` a `SuperRegionMin`. Velikost objektu `RegionMin` tvoří binárně zakódované dynamické položky (2 byty) a pointer na statické položky (na 64-bit platformě 8 bytů). `SuperRegionMin` obsahuje z dynamických položek pouze informaci o vlastníkovi (1 byte). Dále obsahuje pointer na statické položky (8 bytů).

Mapa světa má 42 regionů a 5 super regionů. Tedy počet bytů jednoho stavu

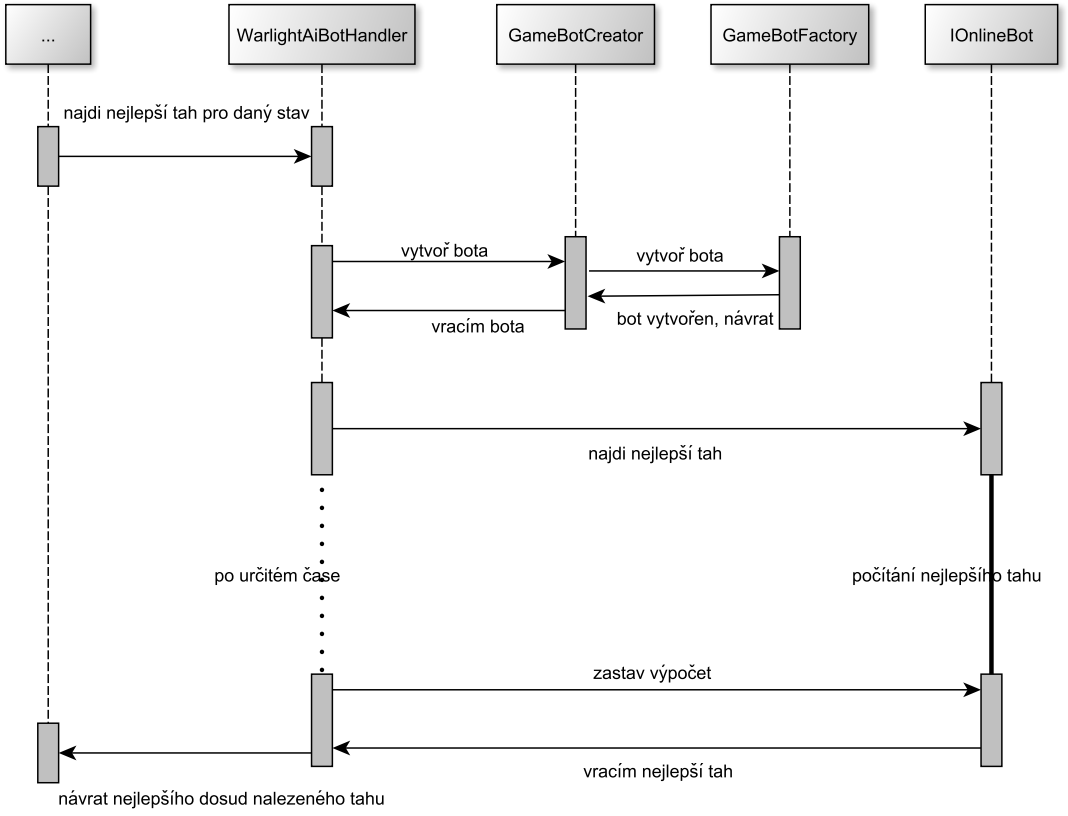
je

$$42 \cdot (2 + 8) + 5 \cdot (1 + 8) = 465$$

. Do 1 GB paměti se vejde přibližně 2,1 milionu těchto stavů.

3.2.3 Objektový návrh

Cílem této sekce je představit nejdůležitější objekty a jak jsou spolu funkčně provázány. Tyto objekty jsou `IOnlineBot`, `GameBotFactory`, `GameBotCreator`, `WarlightAiBotHandler`. `IOnlineBot` je rozhraní, které musí implementovat každý *online* bot. To je takový bot, který dokáže vrátit nejlepší dosud nalezený tah v libovolnou chvíli během výpočtu. `GameBotFactory` je abstraktní předek všech tříd starajících se o konstrukci botů. `GameBotCreator` je třída, která ví, kterou správnou factory třídu vytvořit a díky ní instanciuje bota. `WarlightAiBotHandler` pak pomocí všech výše zmíněných tříd nebo rozhraní vytváří bota a hledá jeho nejlepší tah pro danou pozici.

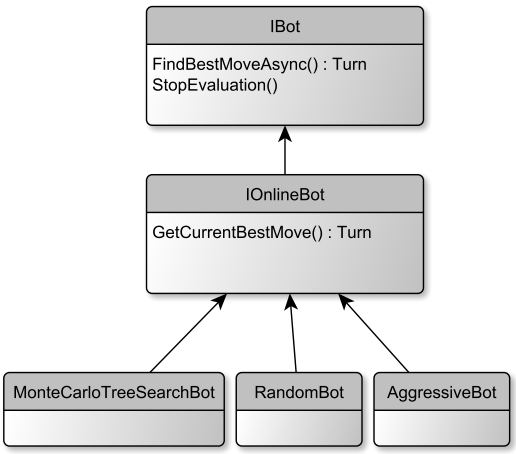


Obrázek 3.12: Interakce hlavních objektů umělé inteligence

Na obrázku je popsána interakce objektů `IOnlineBot`, `GameBotFactory`, `GameBotCreator`, `WarlightAiBotHandler`. Tyto třídy se starají vytvoření, spuštění a zastavení bota. Obrázek demonstruje, jak probíhá komunikace těchto jednotlivých komponent.

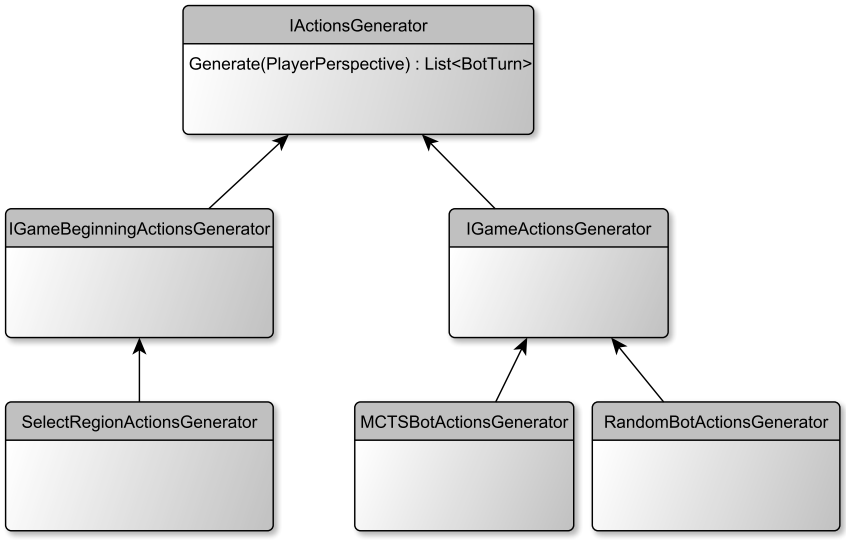
3.2.4 Objektový návrh bota

Hlavním cílem návrhu bota je umožnit jednoduché přidávání dalších implementací. Proto definujeme rozhraní, za které všechny implementace schováme. Rozhraní `IBot` musí splňovat každý bot – schopnost najít asynchronně nejlepší tah a zastavit výpočet v libovolný časový okamžik. V naší práci je však používána místo tohoto základního bota jeho nadstavba – online bot reprezentovaný rozhraním `IOnlineBot`. Výpočet tohoto bota lze také přerušit v libovolný časový okamžik, navíc však garantuje vrácení dosud nejlepšího nalezeného tahu (`IBot` vrácení dosud nejlepšího tahu po přerušení negarantuje). Toto rozhraní pak implementuje například `MonteCarloTreeSearchBot`, `RandomBot` nebo `AggressiveBot`.



Obrázek 3.13: Objektový návrh bota

Bot typicky generuje množinu tahů (neboli akcí), ze kterých se snaží vybrat ten nejlepší. Pro tento účel slouží rozhraní `IActionsGenerator` s jedinou metodou `Generate` vracející seznam dostupných tahů pro daný stav hry a hráče. Tento návrh dovoluje různým botům využívat stejné generátory akcí. Rozlišujeme dva typy těchto generátorů podle fáze hry: `IGameBeginningActionsGenerator` a `IGameActionsGenerator`. Smyslem prvního je určit množinu vhodných obsazení regionů na začátku hry, smyslem druhého je pak nalézt co nejlepší tahy v libovolné další fázi hry.



Obrázek 3.14: Návrh generátorů akcí

Pozn.: `PlayerPerspective` *je stav hry obohacený od Id hráče, z jehož pohledu se tento stav pozorujeme.*

Některé generátory akcí využívají také generátory deploy akcí a generátory attack akcí, což jsou třídy implementující rozhraní `IDeployingActionsGenerator` a `IAttackingActionsGenerator`. Ty umožňují v daném stavu hry vygenerovat seznam deploy, resp. attack akcí. Myšlenkou je umožnit různým generátorům akcí používat stejné generátory deploy či attack akcí (pomocí abstrahování této logiky do speciální třídy, deploy či attack generátoru) a zároveň dovolit různé implementace pro různé tyto generátory (díky schování implementace za interface).

3.2.5 Tovární třídy bota

Vytváření instance bota je poměrně složitá záležitost. Typický bot potřebuje generátory akcí, ty zase potřebují ohodnocovací funkce, které mají mnoho parametrů. Objekty, které bot přijímá parametrem, nazveme konfigurací. Abychom oddělili konfigurování od logiky ostatních tříd, zavedeme si třídy nové – *tovární třídy bota*.

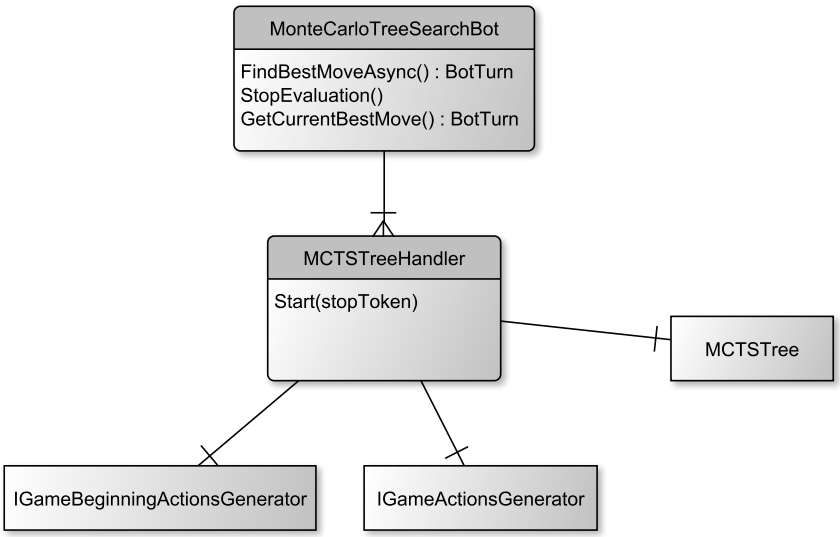
Cílem továrních tříd bota je dodat konfiguraci a vytvořit instanci bota. Tovární třídy dědí od `GameBotFactory`. Ty musí implementovat metodu `CreateInstance`, která na základě stavu hry vrátí vytvořenou instanci bota. V této metodě se děje veškerá jeho konfigurace od volby akčních generátorů až po nastavení koeficientů ohodnocovacích funkcí.

3.2.6 Implementace Monte Carlo tree search bota

Hlavní třída reprezentující Monte Carlo tree search bota je třída `MonteCarloTreeSearchBot`. Ta implementuje interface `IOnlineBot`.

Třída `MonteCarloTreeSearchBot` je navržena tak, aby umožnila paralelizaci metodou Kořenové paralelizace ze sekce 3.1.7. Proto má seznam `MCTSTreeHandler`ů,

které reprezentují výpočet právě na jednom stromu. V závislosti na počtu vláken vytvoří několik instancí třídy `MCTSTreeHandler`, na kterých je metodou `FindBestMoveAsync` spuštěno hledání nejlepšího tahu. Po přerušení metodou `StopEvaluation` je zastaven výpočet na každé instanci `MCTSTreeHandler` a je získán nejlepší tah spojením všech těchto stromů metodou `GetCurrentBestMove`.



Obrázek 3.15: Objektový návrh `MonteCarloTreeSearchBot`

`MCTSTreeHandler` implementuje jednovláknovou verzi algoritmu Monte Carlo tree search. Má metodu `StartEvaluation`, která spustí výpočet. Jednotlivé fáze algoritmu jsou implementovány v metodách `SelectBestNode` (selekce), `Expand` (expanze), `Simulate` (simulace) a `Backpropagate` (zpětná propagace). Na generování možných tahů jsou používány generátory `SelectRegionActionsGenerator` (pro začátek hry) a `MCTSBotActionsGenerator` (pro pozdější fáze hry) implementující chování popsané v sekci 3.1.4.

3.3 Experimenty

V sekcích výše jsme popsali umělou inteligenci pro hru Warlight založenou na Monte Carlo tree search algoritmu. V této se podíváme, jak MCTS bot obstojí v experimentech. Všechny experimenty kromě testování paralelizace budou prováděny použitím jednovláknových botů. Protože to je však velmi časově náročné, budou jednovláknové testy prováděny paralelně. To v některých ohledech ohrozí přesnost výsledků (bot s časem deset sekund například v takových podmínkách může hrát jako bot, který má sedm sekund), závěry z nich však budou prováděny tak, aby je tato paralelizace neovlivnila.

3.3.1 Hardware

Výsledky jsou měřeny na počítači s následujícími parametry:

1. *Procesor* - Intel Core i7-7820HQ 2.90 GHz s 4 fyzickým jádry;

2. *Cache* - L1: 256 KB, L2: 1 MB, L3: 8 MB;

3. *RAM* - 8 GB, rychlost 2400 MHz.

3.3.2 Návrh experimentů

Struktura experimentů je navržena tak, aby otestovala, zdali

1. záleží na volbě počátečních regionů,

2. dokáže MCTS bot porazit referenční boty,

3. lze zjednodušit MCTS bota bez toho, abychom snížili jeho výkonnost,

4. se s rostoucím časem na tah ve hře zvyšuje výkonnost bota,

5. paralelizace zvyšuje kvalitu MCTS bota.

Referenční boti

V této sekci nadefinujeme referenční boty, které budeme v rámci testování používat. Každý z těchto botů bude testovat reakci MCTS bota na určitý druh strategie.

Náhodný bot (RandomBot) Provádí deploy náhodně na region takový, že není vnitrozemní. Útočí na slabší nebo srovnatelně silný region z každého našeho regionu s pravděpodobností 50%. Tento bot v naší práci představuje baseline.

Chytrý náhodný bot (SmartRandomBot) Využívá stejný generátor akcí jako MCTS bot. Na rozdíl od něj však nestaví žádný strom, nýbrž náhodně vybere z množiny tahů, které mu generátor akcí nabízí. Hry s tímto botem ověří, zdali výpočet MCTS bota má vůbec smysl.

Agresivní bot (AggressiveBot) Jak již název napovídá, hraje agresivně. kole nejprve postaví armádu na svůj region, který sousedí s regionen s nejvyšší cenou 3.1.4. Poté útočí na všechny sousedy se slabší armádou. Po zaútočení přesune všechny své armády z vnitrozemí do okrajových regionů. Tento bot ověřuje základní schopnost obrany našeho bota.

Bot s jednou velkou armádou (OneBigArmyBot) Cílem bota s jednou velkou armádou je budovat jednu velkou armádu a útočit s ní na nepřítele. V každém kole nejprve postaví všechny dostupné jednotky na svůj nejsilnější region (takový, který má nejvíce jednotek), poté přesune armádu z vnitrozemních regionů. Nakonec zaútočí ze svého nejsilnějšího regionu na sousední cizí nejcennější region dle 3.1.4. Tímto způsobem otestuje schopnost MCTS bota předvídat vzdálenější hrozby.

3.3.3 Výhoda lepších počátečních regionů

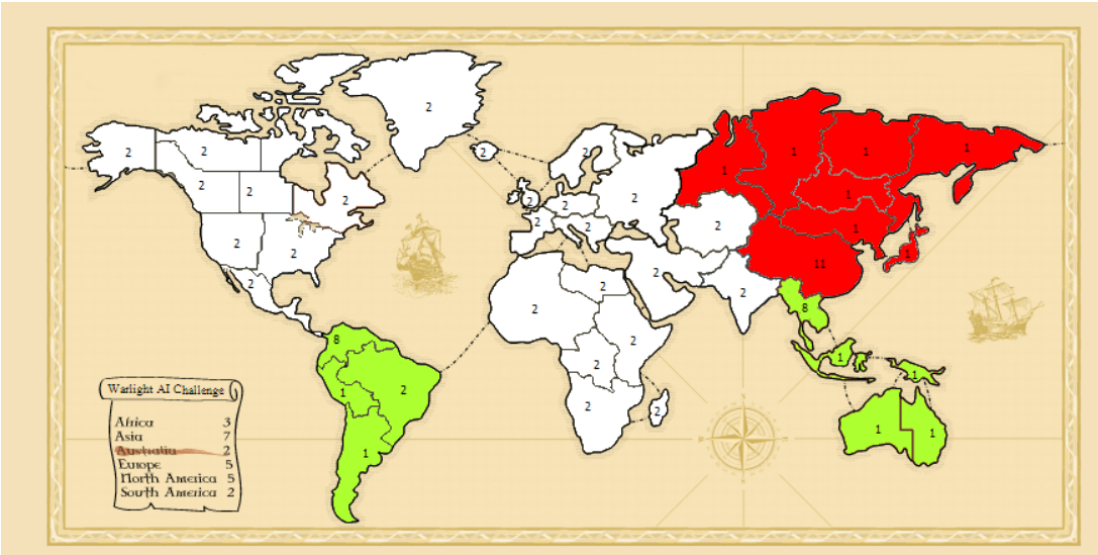
Nejprve se podívejme na hru Monte Carlo tree search bota proti své vlastní verzi, kde ale první z nich bude mít výhodu zvolení lepších regionů na začátku hry. Tím určíme důležitost fáze začátku hry.

První bot bude volit jeden region náhodně z Jižní Ameriky, druhý region pak náhodně z Austrálie. Druhý bot bude volit oba regiony z Asie. Asie je obrovská a je téměř nemožné obsadit ji celou. Naproti tomu Austrálii a Jižní Ameriku lze obsadit rychle a těžit z jejich bonusů již od počátků hry. Bot obsazující Asii tak bude mít nevýhodu. Oběma botům dáme stejný čas na rozmyšlení tahu – 15 sekund. Tento čas by měl být dost velký na to, aby kvůli jeho nedostatku nedocházelo k chybám.

MCTS bot s výhodou (15 s) / soupeři	**MCTS bot** (15 s)
Počet výher	190
Počet proher	10
Úspěšnost	95%

Tabulka 3.2: Statistiky *MCTS bota s výhodou* (15 s) proti *MCTS botovi* (15 s).

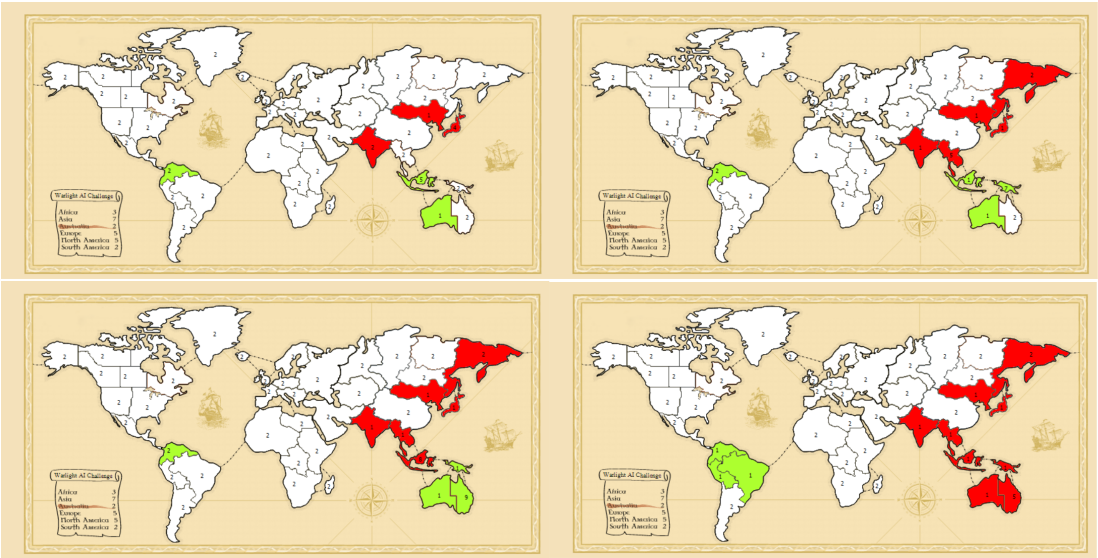
V typické hře bot s výhodou obsadil Jižní Ameriku i Austrálii. V některých hrách se mu nepodařilo obsadit pouze Jižní Ameriku, díky zvýšené produkci jednotek však nakonec dokázal Austrálii dobýt a soupeře převálcovat.



Obrázek 3.16: Typická hra *MCTS bota s výhodou* (zelený) proti znevýhodněnému *MCTS botovi* (červený)

Zelený hráč dokázal obsadit super regiony, zatímco červený ne kvůli velikosti Asie.

Hry, kdy bot s výhodou prohrál, obvykle probíhaly tak, že se jeho soupeři podařilo obsadit Austrálii, zatímco favorit nedokázal získat dostatečně rychle protivýhodu na opačné straně mapy (obrázek 3.17).



Obrázek 3.17: Hráč s výhodou (zelený) proti hráči s nevýhodou (červený)

Červený hráč obsadil Austrálii brzy, zatímco zelený k tomu nezískal žádnou protiváhu. To vedlo k prohře zeleného.

Z výsledků plyne, že bot se lepšími počátečními regiony má znatelnou výhodu. Na počáteční volbě regionů zřejmě nejen záleží, ale dokonce může být klíčovým faktorem ovlivňujícím výsledek.

Ohodnocení regionů na počátku hry je implementováno tak, aby považovalo za cennější regiony spíše menší super regiony. Upřednostňuje tak právě Jižní Ameriku a Austrálii, které je jednoduché na počátku hry celé obsadit.

Abychom odstínili další experimenty významu volby regionu na začátku hry, budeme je volit zcela náhodně. To by mělo, při dostatečném počtu her, zajistit jak variabilitu her, tak konvergenci k výsledku nezávislému na počáteční volbě.

3.3.4 Hry s referenčními boty

Podívejme se nyní na hry Monte Carlo tree search bota proti referenčním botům – proti náhodnému botovi, chytrému náhodnému botovi, agresivnímu botovi a botovi s jednou velkou armádou.

Metrika / MCTS bot	**MCTS bot (10 s)**
Prům. list s min hloubkou	4,29
Prům. list s max hloubkou	6,36
Med. # simulací	457
Med. # vrcholů	2986

Tabulka 3.3: Metriky *MCTS bota* (10 s).

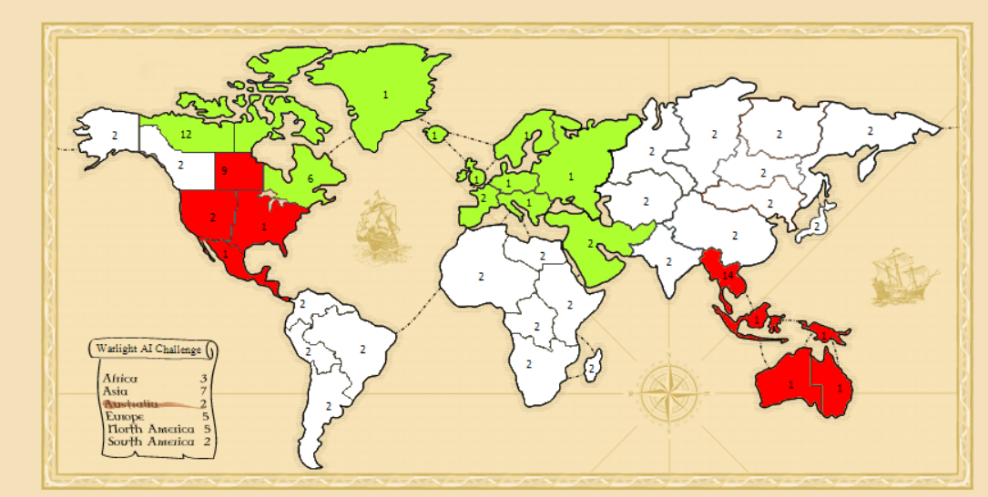
MCTS/soupeři	**Náhodný bot**	**Chytrý náhodný bot**	**Agresivní bot**
Počet výher	500	481	436
Počet proher	0	19	64
Úspěšnost	100%	96,2%	87.2%

Tabulka 3.4: Statistiky *MCTS bot* (10 s) proti *náhodnému botovi*, *chytrému náhodnému botovi* a *agresivnímu botovi*.

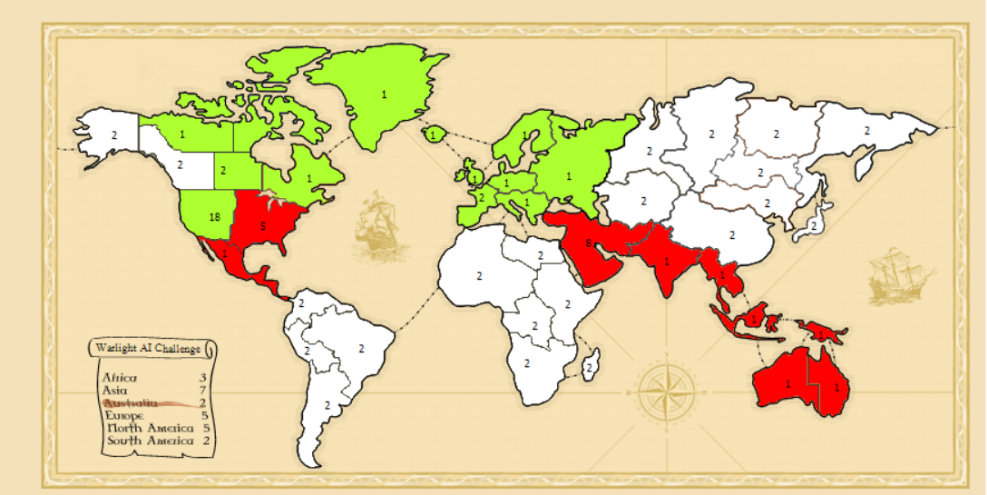
Příčina mizerného výsledku *náhodného bota* je zřejmá. Z analýzy (3.1.1) víme, že větvící faktor je nesmírně velký a počet rozumných tahů malý. Volíme-li tah náhodně, pravděpodobnost zvolení rozumného je nízká.

Chytrý náhodný bot dopadl jen o trochu lépe než náhodný bot. Tento bot volí svůj tah ze stejné množiny, ze které by v jeho situaci volil MCTS bot, na rozdíl od něj volí však náhodně. Skóre ve prospěch MCTS bota ukazuje, že jeho výpočet má přínos a je lepší než náhodná volba.

Agresivní bot skončil ze všech nejlépe. Ve hrách, kde zvítězil, měl buď štěstí, nebo využil trhliny v obraně MCTS bota. Ten bere v úvahu obranný deploy až ve chvíli, kdy je nepřítel na hranici jeho území. To je problém v situacích, kdy na ně přijde nepřátelská armáda tak velká, že jedno kolo deploy fáze na ubránění nestačí. Další trhlina v obraně je ohodnocovací funkce. Bot se snaží prioritně bránit cenné regiony. Jako cenný však nedokáže rozpoznat region sousedící s jinými cennými regiony. Ohodnocovací funkce by v budoucnu takto šla zlepšit.



(a) Situace, kdy měl zelený rozpoznat hrozbu a začít bránit Evropu v jeho nejvýchodnějším regionu.



(b) Červený dobyl klíčový region a nyní může zaútočit na dva regiony Evropy, které zelený nedokáže ubránit.

Obrázek 3.18: *MCTS bot* (zelený) proti *agresivnímu botovi* (červený).

Na obrázku 3.18 vidíme právě tuto situaci. MCTS bot nerozpoznal včas význam obrany regionu, a proto o něj přišel. Jeho soupeř tak získal cestu do Evropy, která mu dala důležitý bonus v produkci jednotek a dokázal tak zvítězit.

MCTS bot vyhrává naopak na schopnost bránit a vyčkávat se svým útokem. To způsobuje menší ztráty v bojích a získává mu dlouhodobou výhodu.

Z rozboru her s agresivním botem plyne, že MCTS bot má problémy s předvídáním vzdálenějších hrozeb. Zkusíme tento problém zneužít *botem s jednou velkou armádou (OneBigArmyBot)*.

MCTS/soupeři	**Bot s jednou velkou armádou**
Počet výher	497
Počet proher	3
Úspěšnost	99,4%

Tabulka 3.5: Statistiky *MCTS bota* (10 s) proti *botovi s jednou velkou armádou*.

Výsledky ukazují, že *bot s jednou velkou armádou* byl drtivě poražen. Jak je v některých situacích výhodou, že tento druh bota koncentruje veškerou armádu do jednoho místa, tak ve většině situací to je naopak nevýhoda. Obvykle tak sice vybuduje velkou armádu a prochází dlouho bez odporu, nakonec je ale zastaven, protože MCTS bot mezitím obsadil celý zbytek mapy, a díky větší produkci postavil armádu ještě silnější.

Obrázek 3.19: Typická hra *MCTS bota* (zelený) proti *botovi s jednou velkou armádou* (červený)

Červený sice má jednu velkou armádu, ale zelený ovládá většinu mapy. Díky tomu má vyšší produkci a nakonec armádu červeného porazí.

3.3.5 MCTS bot bez obrany

Monte Carlo tree search bot kombinuje tři strategie: útočnou, expanzivní a obrannou. Útočnou i expanzivní nutně potřebuje k fungování. Pokud nebude útočit, nikdy nezaútočí na nepřítele a nemůže tedy vyhrát. Pokud nebude expandovat, nikdy neobsadí veolný region. Nebude se tedy ani rozvíjet, protože se obvykle na začátku hry obsazují nejprve volné regiony. Obranná strategie je formována schopností bránit nejohroženější regiony a vyčkávat s útokem. V této sekci otestujeme, zdali ji MCTS bot skutečně potřebuje. Necháme hrát MCTS bota proti *MCTS botovi bez obrany (NoDefenseMctsBot)*.

MCTS bot (15 s) / soupeři	**MCTS bot bez obranny (15 s)**
Počet výher	165
Počet proher	35
Úspěšnost	82,5%

Tabulka 3.6: Statistiky *MCTS bot* (15 s) proti *MCTS botovi bez obrany* (15 s).

MCTS bot bez obrany byl ve většině her poražen. Z pravidel víme, že bránící hráč má výhodu očekávaně menších ztrát (sekce 1.5.2). MCTS bot bez obrany

tak v boji přichází o více jednotek, což se negativně odrazilo na jeho úspěšnosti.

3.3.6 Vliv doby výpočtu na výkonnost

Drtivou porážkou chytrého náhodného bota jsme ukázali, že výpočet MCTS bota má smysl. V této sekci na tento fakt navážeme a prozkoumáme vliv doby výpočtu na výkonnost MCTS bota.

MCTS (10 s)/soupeři	**MCTS (2 s)**	**MCTS (5 s)**
Počet výher	274	168
Počet proher	126	132
Úspěšnost	68,5%	56%

Tabulka 3.7: Statistiky *MCTS bota* (10 s) proti botům *MCTS* (2 s) a *MCTS* (5 s).

MCTS (20 s)/soupeři	**MCTS (5 s)**
Počet výher	358
Počet proher	242
Úspěšnost	59,67%

Tabulka 3.8: Statistiky *MCTS bot* (20 s) proti *MCTS botovi* (5 s).

Výsledky dokazují závislost času na úspěšnosti bota. Čím více času má MCTS bot, tím lépe hraje.

3.3.7 Paralelizace

V sekci 3.1.7 byla představena paralelizace algoritmu Monte Carlo tree search. Zvolena byla tzv. kořenová paralelizace. Nyní ověříme, zdali ve Warlightu bude také mít dle očekávání vyšší výkonnost.

V testovacích hrách přidělíme každému botovi na tah 15 sekund a necháme vždy hrát jednovláknového bota proti jeho vícevláknové variantě. Aby byl výpočet co nejpřesnější, bude použito maximálně tolik vláken, kolik je fyzických jader na počítači. I když je každé fyzické jádro tvořeno dvěma logickými, při použití obou dochází ke snížení výkonu.

MCTS/ PMCTS	**2 vlákna**	**4 vlákna**
Počet výher	52	85
Počet proher	48	65
Úspěšnost	52%	56,7%

Tabulka 3.9: *MCTS bot (MCTS)* proti *paralelním MCTS botům (PMCTS)* (2 vlákna, 4 vlákna). Každý bot má na tah 15 sekund.

V práci Chaslot a kol. [3] použitím kořenové paralelizace vzrostla výkonnost vícevláknového bota více než dvojnásobně (tabulka 3.1). Naše měření však došlo k diametrálně odlišným výsledkům. Pro zjištění příčiny se zkusíme podívat na některé metriky stavěných MCTS stromů.

Metrika / MCTS bot (15 s)	**1 vlákno**	**2 vlákna**	**4 vlákna**
Med. # simulací	1272	1096	836
Med. # vrcholů	10344,5	9164,5	5841

Tabulka 3.10: Metriky jednotlivých stromů (počítáno individuálně pro každý strom) *MCTS botů* (15 s) při různém počtu použitých vláken.

Na tabulce 3.10 vidíme, že se zvyšujícím se počtem vláken dochází ke snížení výkonu jednotlivých vláken. Toto chování je zvláštní vzhledem k tomu, že výpočet všech vláken je zcela oddělený a neobsahuje žádné kritické sekce. Příčin může být několik. Nás napadlo použití *Intel Turbo Boost*[1] během testování a problém čtení více vláken ze společné paměti. Na vyvrácení první zmíněné jsme zkusili *Turbo Boost* vypnout a spustit výpočty znovu. Procesor se přestal přetaktovávat, nicméně výpočetní výkon vláken paralelního bota byl stále slabší. Druhá zmíněná možná příčina se obtížně zkoumá a nebyla proto prověřena.

Nižší počet simulací a vrcholů zřejmě vede ke zhoršení odhadu pozice a snížení výkonnosti paralelních botů. Protože je však výpočet kola nedeterministický, více stromů dokáže lépe prozkoumat různá pokračování hry a odhadnout skutečnou kvalitu stavu. To je výhodou paralelních MCTS botů. Věříme, že kdyby více spuštěných vláken v jednu chvíli nezhoršovalo jejich individuální výkonnost, paralelní MCTS bot by prokazoval lepší výsledky než jednovláknový.

Závěr

V práci jsme připravili grafické rozhraní pro vývoj umělé inteligence do hry Warlight. To obsahuje simulátor, který zobrazuje spuštěnou hru dvou botů, a možnost hry jednoho hráče proti počítači. Navíc byla přidána možnost hry proti jiným lidským hráčům. Dále jsme naimplementovali bota využívajícího Monte Carlo tree search algoritmus, jeho vícevláknovou verzi a několik jednoduchých referenčních botů. Provedli jsme experimenty, kde MCTS bot ukázal svou převahu a porazil ostatní referenční boty. Také z nich však vyplynulo, že vícevláknový MCTS bot při použití tzv. kořenové paralelizace na architektuře osobního počítače, na kterém byly prováděny experimenty, není lepší než bot jednovláknový.

3.4 Budoucí práce

3.4.1 Generování akcí

Ačkoliv MCTS bot hraje poměrně dobře, experimenty odhalily mezery především v obranné strategii. Jedna možnost je dopsat chytřejší generátor deploy akcí. Druhou obtížnější možností je navrhnout zcela nový generátor akcí, jehož deploy a attack akce budou generovány spolu a tyto strategie tak nebudou od sebe odtrženy.

3.4.2 Ohodnocovací funkce

Výkon bota stojí a padá s funkcemi, které určuje kvalitu jednotlivých stavů nebo herních objektů. Parametry této funkce byly nastaveny ručně na základě našeho odhadu. Tyto parametry by šly zpřesnit například pomocí evolučního algoritmu. Spolu s parametry by mohla ohodnocovací funkce brát v úvahu hodnoty sousedních regionů.

3.4.3 Paralelní Monte Carlo tree search

Měření ukázala, že paralelní Monte Carlo tree search z neznámé příčiny nepřináší očekávanou výhodu oproti jeho jednovláknové variantě. Možné navázání je zkusit najít a opravit tuto příčinu. Další možností je naimplementovat vícevláknového bota použitím přístupu, který nebude tolik spoléhat na individuální výkon vláken. Z přístupů uvedených v sekci 3.1.7 se zdá být nejvhodnější metoda používající lokální zámky a koncept virtuální prohry.

Literatura

[1] (2008). Intel® turbo boost technology in intel® core™ microarchitecture (nehalem) based processors.

[2] Benjamin628. Learning warlight strategy. URL https://docs.google.com/document/d/1ERn5UL2-K6d5VUmK8zkWA1j1m1GIqPUiHCYmbWBufw4/edit.

[3] Chaslot, G. M., Winands, M. H. a Herik, H. J. (2008). *Parallel Monte-Carlo Tree Search*. CG '08. Springer-Verlag, Berlin, Heidelberg. ISBN 978-3-540-87607-6. doi: 10.1007/978-3-540-87608-3_6. URL http://dx.doi.org/10.1007/978-3-540-87608-3_6.

[4] GG (2013). Gg warlight strategy guide. URL https://drive.google.com/file/d/0B024HXqzvoGCejhhajNFa1ZOVlE/edit.

[5] Kocsis, L. a Szepesvári, C. (2006). *Bandit Based Monte-carlo Planning*. ECML'06. Springer-Verlag, Berlin, Heidelberg. ISBN 3-540-45375-X, 978-3-540-45375-8. doi: 10.1007/11871842_29. URL http://dx.doi.org/10.1007/11871842_29.

[6] Norman. M'hunters warlight strategy guide. URL https://docs.google.com/document/d/1NyhCpIQKShAbWGXicO_ph9whV_UyMS-3a7_re0C7R8Y/edit.

[7] Riddles.io. Warlight ai challenge. URL http://theaigames.com/competitions/warlight-ai-challenge.

[8] Wikipedia. Alfa-beta ořezávání. URL https://upload.wikimedia.org/wikipedia/commons/thumb/9/91/AB_pruning.svg/800px-AB_pruning.svg.png.

[9] Wikipedia. Monte carlo tree search fáze. URL https://upload.wikimedia.org/wikipedia/commons/thumb/6/62/MCTS_%28English%29_-_Updated_2017-11-19.svg/808px-MCTS_%28English%29_-_Updated_2017-11-19.svg.png.

Seznam obrázků

Seznam tabulek

A. Přílohy

A.1 Obsah přiloženého CD

- `/Bin` - binární soubory programu
- `/Documentation/UserDocumentation/doc.html` - uživatelská dokumentace
- `/Documentation/ProgrammerDocumentation/doc.html` - programátorská dokumentace
- `/Documentation/ProgrammerDocumentation/Generated/index.html` - vygenerovaná dokumentace ze zdrojových kódů
- `/Experiments/experiments.zip` - záznamy z experimentů
- `/Source` - zdrojové kódy
- `/Text/prace.pdf` - text práce