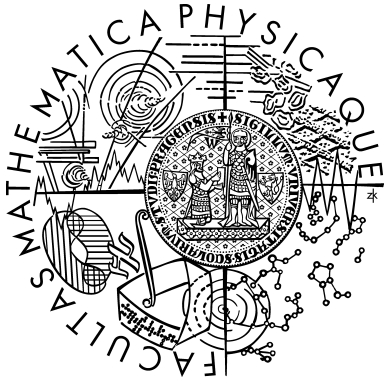
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Tibor Blénessy

Prostředí pro vývoj a testování botů, umělá inteligence se schopností učení

Ústav formální a aplikované lingvistiky

Vedoucí bakalářské práce: Mgr. Miroslav Spousta
Studijní program: Obecná informatika

2007

Na tomto mieste by som sa chcel poďakovať Miroslavovi Spoustovi za vedenie bakalárskej práce a za jeho veľmi príjemný a ústretový prístup. Ďalej by som sa chcel poďakovať svojím rodičom za podporu pri písaní práce.

Prehlasujem, že som bakalársku prácu napísal samostatne a výhradne s použitím citovaných prameňov. Súhlasím so zapožičiavaním práce a jej zverejňovaním.

V Prahe dňa 10. augusta 2007 Tibor Blénessy

Obsah

Název práce: Prostředí pro vývoj a testování botů, umělá inteligence se schopností učení
Autor: Tibor Blénessy
Katedra (ústav): Ústav formální a aplikované lingvistiky
Vedoucí bakalářské práce: Mgr. Miroslav Spousta
E-mail vedoucího: Miroslav.Spousta@mff.cuni.cz

Abstrakt: V předložené práci se zabývam návrhem a implementací herního prostředí pro počítačem řízené hráče ve 2D prostoru. Další částí práce je návrh a implementace hráče ve vytvořeném prostředí. Pri návrhu chování hráče se používa algoritmus A*, stejný algoritmus je použit také pro řízení pohybu hráče po mapě. K vylepšení heuristické funkce algoritmu A* je použita technika genetických algoritmů. Projekt je implementován na platformě Java SE 5.

Klíčová slova: umělá inteligence, algoritmus A*, genetické algoritmy, hry

Title: Software robot development and testing environment with AI adaptability features
Author: Tibor Blénessy
Department: Institute of Formal and Applied Linguistics
Supervisor: Mgr. Miroslav Spousta
Supervisor's e-mail address: Miroslav.Spousta@mff.cuni.cz

Abstract: In the presented work I devote to design and implementation of game enviroment for computer driven players in 2D space. Next I present design and implementation of player to the designed enviroment. In player design I have used A* algorithm for action planning. I have used A* also for movement managment. I have used genetic algorithms to enhance heuristic function of A* algorithm. Project is implemented on Java SE 5 platform.

Keywords: artifficial inteligence, A* algorithm, genetic algorithms, games

Kapitola 1

Úvod

1.1 Umelá inteligencia

Pod pojmom umelá inteligencia sa skrýva mnoho rôznych vedných disciplín. Vo všeobecnosti možno povedať, že umelá inteligencia sa ako vedný odbor snaží nájsť také postupy riešenia problémov, akoby tieto problémy riešila inteligentná bytosť. Pričom za inteligentné bytosti sa neskromne považujeme my – ľudia. Cieľom je potom tieto postupy algoritmizovať tak, aby ich bol schopný vykonávať stroj, väčšinou vo forme programu.

Medzi typické úlohy umelej inteligencie patrí rozpoznávanie obrazu, rozpoznávanie reči, data mining, riadenie a plánovanie,... a mnohé ďalšie. Tieto úlohy su charakteristické tým, že buď ich nie je možno vyriešiť priamo analyticky alebo by analytické riešenie zabralo príliš mnoho času. Práve preto nastupujú postupy, ktoré vedia nájsť aspoň približné riešenie problému v rozumnom čase.

Na riešenie týchto úloh sa používa množstvo rôznych techník. Uvediem ako príklad aspoň niektoré podľa [2]:

Konečné automaty sú systémy založené na stavoch. Pre každý stav je definovaná prechodová funkcia, ktorá určuje za akých podmienok sa stav zmení na iný stav. Konečný automat sa vždy nachádza práve v jednom stave, jeho práca skončí, ak sa dostane do niektorého z koncových stavov.

Neurónové siete sú inšpirované nervovými sústavami živých organizmov. Základným prvkom je neurón, ktorý je synapsami spojený s ostatnými neurónmi v sieti. Úpravou vnútorných numerických parametrov je potom sieť neurónov schopná abstrahovať vzťahy medzi vstupmi a požadovanými výstupmi. Používajú sa napr. pri rozpoznávaní obrazu.

Genetické algorimy sú takisto inšpirované prírodou. Sú vhodné na vyhľadanie optimálnych hodnôt. Najprv sa tieto hodnoty vygenerujú náhodne a vytvoria tzv. populáciu, z ktorej potom operáciami kríženia a náhodných mutácií vytvoríme ďalšiu generáciu. Tento postup sa iteruje až kým populácia nevyhovuje zadaným požiadavkám.

Fuzzy logika na rozdiel od Boolovskej logiky nepoužíva na vyjadrenie pravdivosti dve hodnoty (pravda, nepravda), ale reálne čísla, ktoré vyjad-

rujú mieru pravdivosti formulí. To jej umožňuje lepšie simulovať vlastnosti reálneho sveta.

Vyhľadávacie metódy sa zaoberajú vyhľadávaním sledov akcií alebo stavov v grafe, ktoré vedú k určenému cieľu alebo maximalizujú určenú hodnotu.

V predloženej práci som na implementáciu počítačom riadeného hráča využil vyhľadávacie metódy – konkrétne algoritmus A*. Tento algoritmus používa na ohodnotenie stavov heuristickú funkciu, ktorej parametre som potom skúšal optimalizovať pomocou genetických algoritmov.

Umelá inteligencia v počítačových hrách

V počítačových hrách nachádza umelá inteligencia široké uplatnenie. Medzi jej hlavné úlohy patrí modelovanie herného sveta a riadenie počítačom kontrolovaných postáv. Postavy v hre sa musia správať tak, aby pôsobili logicky a vierohodne. Cieľom je poskytnutie čo najlepšieho herného zážitku.

Prístupy k programovaniu UI v počítačových hrách

Podľa [1] možno prístup k programovaniu umelej inteligencie v počítačových rozdeliť na dva druhy:

Tradičný prístup

Umelá inteligencia v hre je súčasťou hernej logiky a má prístup ku všetkým informáciám o hre. Pohyb a ostatné činnosti UI sú robené priamo v hlavnom cykle herného engine. Tento spôsob sa používa od počiatku počítačových hier a ako taký dosahuje dobré výsledky. Programovanie umelej inteligencie však môže byť náročné a zdĺhavé.

Moderný prístup

Moderný prístup je založený na agentoch. Agent je samostatná entita v hre, ktorá má jasne definované rozhranie ako môže s hrou interagovať. Toto oddelenie prináša mnoho výhod – vývoj agenta môže prebiehať oddelene, jeho správanie viac zodpovedá skutočnosti. Ďalšou dôležitou vlastnosťou agenta je učenie. Vývojár už nevytvára správanie „natvrdo“, ale agent je schopný sám inteligentne reagovať na rôzne situácie.

1.2 Cieľe práce

Cieľom tejto práce bolo vytvoriť jednoduché herné prostredie, v ktorom by sa dali ľahko implementovať a testovať počítačom riadený hráči (označovaný tiež termínom *bot*).

Ďalším cieľom bolo vytvorenie bota do tohto prostredia, ktorý by bol na základe predošlých hier schopný zlepšiť svoje parametre.

Kapitola 2

Herné prostredie

2.1 Popis herného prostredia

Pravidlá

Pravidlá a prostredie hry sú inšpirované hrou Tunneler. Hráči sú tanky, ktoré bojujú proti sebe. Cieľom bota je získať najväčšie skóre. Skóre sa získava za zásah a zničenie súpera a v menšej miere za pohyb po dlaždici typu *SOIL*.

Mapa Boti sa pohybujú na dvojrozmernej mape. Mapa sa skladá z dlaždíc, ktoré tvoria osem–súvislú oblasť. Každý objekt hry sa môže vyskytovať na práve jednej dlaždici. Objekt môže mať na dlaždici rôzny smer natočenia, ktorý potom určuje smer jeho pohybu. Dlaždice môžu byť rôznych typov, každá má určenú cenu pohybu. Na základe akcie botov sa dlaždice môžu zmeniť. Každý bot pri pohybe na mape zanecháva stopu (*ROAD*), pričom cena pohybu po stope je menšia ako cena pohybu mimo stopy. Jednotlivé typy dlaždíc sú uvedené v tabuľke 2.1.

Ťah Ťah predstavuje reakciu bota na prostredie. Možné ťahy bota sú uvedené v tabuľke 2.2.

Dlaždica	**Význam**
SOIL	vysoká cena pohybu
ROAD	vzniká pohybom po *SOIL*, nízka cena pohybu
BASE	základňa, na tejto dlaždici sa dobíja hodnota štítu a paliva
BASEWALL	hradba, nepriechodná dlaždica
ROCK	nepriechodná dlaždica
OUTOFMAP	dlaždica na pozícii mimo mapy, nepriechodná
UNKNOWN	dlaždica používaná interne hráčom na vyjadrenie neznámej pozície

Tabuľka 2.1: Typy dlaždíc na mape

Ťah	Význam ťahu
NONE	žiadny pohyb
FORWARD	pohyb dopredu
BACKWARD	pohyb dozadu
LEFT	zmena natočenia v protismere hodinových ručičiek
RIGHT	zmena natočenia v smere hodinových ručičiek
shooting	streľba. Streľba je nezávislá od pohybu. Strieľa sa vždy v smere natočenia

Tabuľka 2.2: Ťahy bota v hernom prostredí

Bot Každý bot má dva základné parametre – palivo a štít. Palivo sa míňa pohybom. Štít sa míňa pri zásahu strelou iného bota. Palivo aj štít sa dá doplniť na základni.

Základňa Základňa je špeciálny typ dlaždice, na ktorom sa botovi dopĺňa palivo a štít. Pre každého bota existuje „domovská“ základňa, na ktorej sa objaví po svojom zostrelení.

Pravidlá sa dajú konfigurovať viacerými parametrami. Tieto sú popísané v programátorskej dokumentácii projektu.

2.2 Architektúra herného prostredia

Pri návrhu herného prostredia som kládol najväčší dôraz na jeho rozšíriteľnosť. Preto je prísne rozdelený návrh a jeho implementácia. V návrhu som použil viaceré návrhové vzory podľa [5].

V ďalšom texte na lepšie dokumentovanie používam UML diagramy. Tieto diagramy pre prehľadnosť nie sú úplné. Úplný popis jednotlivých tried možno nájsť v programátorskej dokumentácii.

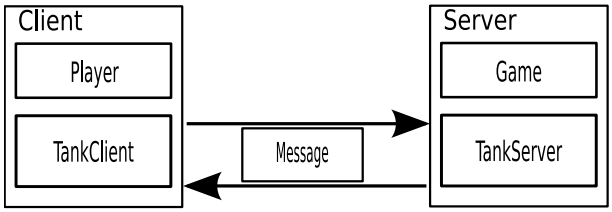
Celková architektúra

Celková architektúra herného prostredia je znázornená na obrázku 2.1. Prostredie používa architektúru klient/server, pričom hra beží na serveri a jednotlivý hráči sa ku nej pripájajú ako klienti. Prostredie je tvorené štyrmi časťami: hráčom, klientskou časťou komunikácie, serverovou časťou komunikácie a herným enginom.

Rozdelenie prostredia na jednotlivé komponenty zjednodušuje ich vývoj, keďže vzťahy medzi nimi su určené pomocou dobre definovaných rozhraní. Použitie archiktektúry klient/server prináša výhodu v možnosti spúštania klientov po sieti. Nevýhodou je zložitejšia výmena informácií medzi hrou a problémy spojené so synchronizáciou.

Komunikácia

Komunikácia medzi serverom a klientami priebieha pomocou správ.



Obr. 2.1: Diagram celkovej architektúry herného prostredia

Prijímateľ správ je definovaný rozhraním *MessageListener*. Každý prijímateľ správ vlastní objekt, ktorý správy obsluhuje. Tento objekt je určený pomocou rozhrania *MessageHandler*, ktorý obsahuje metódy na obsluhu každého druhu správy. Rozhranie *MessageHandler* je implementovaný abstraktnou triedou *DefaultMessageHandler*, ktorá implementuje každú metódu prázdnym telom.

Komunikačná vrstva na strane klienta (*TankClient*) a servera (*TankServer*) rozširuje túto triedu a preťažuje metódy podľa správ o ktorých obsluhu má záujem.

Odosieľateľ správ je definovaný pomocou rozhrania *MessageSender*, ktorý obsahuje jedinú metódu *sendMessage* na zaslanie správy.

Tieto rozhrania kompletne definujú príjimanie a odosielanie správ. Keďže hra môže prebiehať po sieti, je potrebná ďalšia vrstva medzi rozhrania *MessageSender* a *MessageListener*, ktorá zabezpečí zakodóvanie správy a jej zaslanie na strane odosieľateľa, prijatie a jej dekódovanie na strane prijímateľa.

Sieťová vrstva Pre posielanie správ po sieti je potrebné zakódovať správu na prúd bytov. Aby aplikácia nebola závislá na formáte kódovania je kódovanie každej správy definované cez rozhranie *MessageEncoder*.

Spracovanie prúdu bytov a vytvorenie správy zabezpečuje rozhranie *MessageWorker*. *MessageWorker* používa na dekódovanie správ rozhranie *MessageDecoder*.

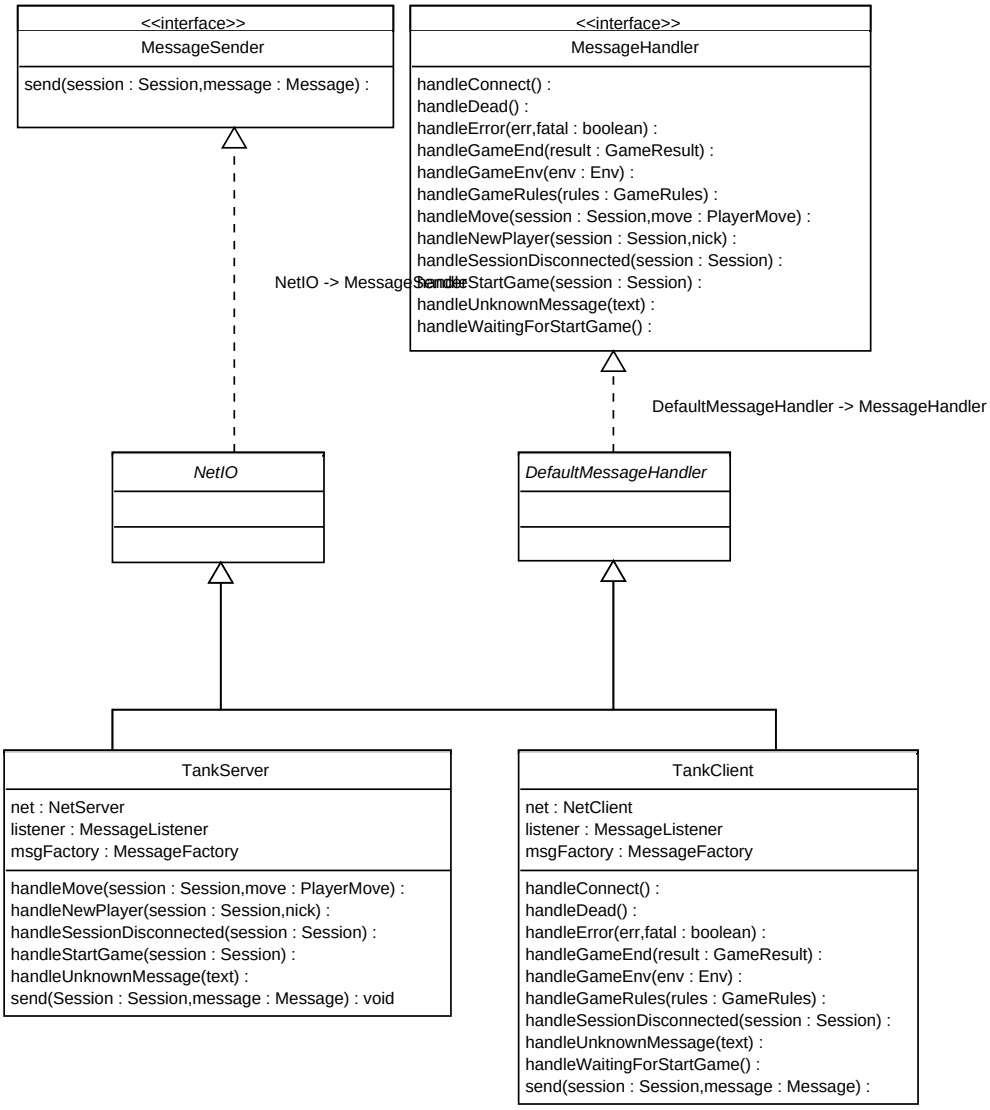
Na vytvorenie objektov celej vrstvy sa používa abstract factory *MessageConfig*. Východzia implementácia všetkých rozhraní sa dá získať pomocou implementácie tejto továrne v triede *DefaultMessageConfig*.

Správy Každá správa je potomkom abstraktnej triedy *Message*. Objekty správ sú navrhnuté pomocou návrhového vzoru visitor. Správa používa dva druhy visitorov – na obsluhu správy sa používa visitor *MessageHandler* a na kódovanie správy sa používa visitor *MessageEncoder*. Aby sa správy dali rozšíriť, na vytváranie objektov správ sa používa trieda *MessageFactory* podľa návrhového vzoru abstract factory. Popis správ je uvedený v tabuľke 2.2.

Dôkladné rozdelenie komunikačnej vrstvy na jednotlivé interface umožňuje veľkú variabilitu pri jej implementácii, zároveň sa však vrstva stáva trochu neprehľadnou. Triedy *TankClient* a *TankServer* preto poskytujú zjednodušenie pohľadu na celú vrstvu (návrhový vzor Facade) tým, že implementujú rozhrania *MessageSender* a *MessageHandler* súčasne.

Názov	Popis
turn	číslo kola
myself	informácie o hráčovi – pozícia, stav štítu a stav paliva
view	mapa viditeľného okolia
figs	zoznam viditeľných objektov hry

Tabuľka 2.3: Informácie zasielané klientovi v správe *GameEnvMessage*



Obr. 2.2: Diagram komunikačnej vrstvy

Správy zasielané klientom	
WaitingForStartGame	Server je pripravený na začiatok hry a čaká, kedy mu prvý pripojený klient pošle *StartGameMessage*
GameEnvMessage	Správa o stave hry pre klienta. Viac v tabuľke 2.3
ErrorMessage	Oznámenie servera o chybe zo strany klienta.
DeadMessage	Oznámenie hráčovi o tom, že bol zostrelený. Podľa nastavenia hry môže v hre pokračovať ďalej – znovu sa objaví na svojej základni.
GameEndMessage	Oznámenie o konci hry. Obsahuje informácie o získanom skóre
Správy zasielané serveru	
MoveMessage	Ťah hráča. Viď tabuľka 2.2
StartGameMessage	Reakcia hráča na správu *WaitingForStartGame*. Správu môže odoslať iba hráč, ktorý sa pripojil ku serveru ako prvý.

Tabuľka 2.4: Správy zasielané medzi serverom a klientom

Herný engine

Simuláciu herného sveta zabezpečuje herný engine. Hra prebieha v jednotlivých kolách. Priebeh jedného kola je znázornený v ukážke 2.3.

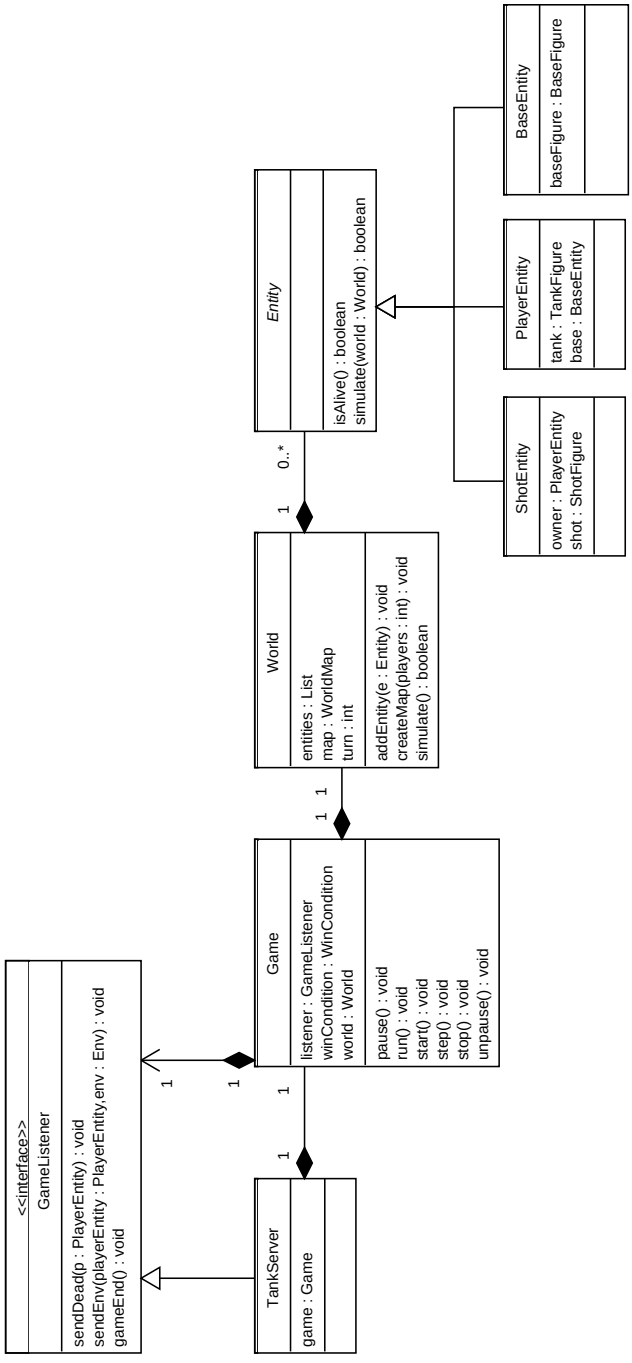
1. Získaj ťahy jednotlivých hráčov
2. Preveď ťah
    2. a Nech E je objekty hry, ktorý ešte v tomto ťahu neťahal
    2. b Preveď ťah pre E
    2. c Opakuj 2. a pre všetky objekty hry
3. Objekty, ktoré počas ťahu „zomreli“ vyraď zo zoznamu objektov
4. Odošli informácie o zmene sveta hráčom

Ukážka 2.3: Pseudokód priebehu jedného kola

Herný engine je implementovaný v triede *Game*. Prevedenie ťahu v hre zabezpečuje trieda *World*, ktorá obsahuje všetky informácie o hernom svete v jednom kole. Každý objekt v hre je reprezentovaný implementáciou abstraktnej triedy *Entity*. Trieda *World* v simulácii ťahu zavolá virtuálnu metódu *simulate* nad každým objektom hry.

Rozhranie medzi triedou *Game* a triedou *TankServer* z komunikačnej vrstvy je definové rozhraním *GameListener*. Toto rozhranie definuje metódy na zaslanie informácii o hre hráčom, zaslanie správy o zabití hráča a metódu na ukončenie behu programu.

Prehľad herného engine je na obrázku 2.4



Obr. 2.4: Diagram herného engine

Hráč

Hráč je definovaný rozhraním *PlayerInterface*. To obsahuje metódy na obsluhu udalostí v hre, initializáciu hráča a ukončenie hráča. Rozhranie je implementované v abstraktnej triede *AbstractPlayer*. Táto poskytuje základnú funkčnosť pre konkrétne implementácie hráčov, ako spustenie hráča vo vlastnom vlákne, zobrazovanie priebehu hry a pod.

Keďže *AbstractPlayer* je spustený v samostatnom vlákne, je potrebné oddeliť zavolanie metód na obsluhu udalostí a reakciu na tieto udalosti. Reakcie na udalosti sú preto definované pomocou rozhrania *PlayableInterface*. Trieda *AbstractPlayer* poskytuje pre toto rozhranie implementáciu s prázdnym telom metód.

Konkrétne implementácie hráčov potom rozširujú triedu *AbstractPlayer* a preťažujú metódy z *PlayableInterface*. Prehľad konkrétnych implementácii je v tabuľke 2.5.

Rozhranie medzi *PlayerInterface* a triedou *TankClient* z komunikačnej vrstvy je definované rozhraním *PlayerListener*. Toto rozhranie obsahuje metódu na zaslanie ťahu a na zastavenie klienta (môže sa použiť na vzdanie hry).

Skriptovanie hráčov Pomocou hráča *ScriptPlayer* je možné implementáciu *PlayableInterface* zabezpečiť pomocou skriptu v skriptovacom jazyku *Beanshell* [3] [1]. Táto implementácia preťažuje metódy z *PlayableInterface* tak, že zavolá príslušnú metódu z poskytnutého skriptu.

Názov triedy	**Popis**
HumanPlayer	Umožňuje ovládať hráča pomocou klávesnice
RandomPlayer	Posiela náhodné ťahy. Implementované kvoli testovaniu
ScriptPlayer	Umožňuje skriptovanie pomocou BeanShell
AIPlayer	Inteligentný hráč ovládaný počítačom

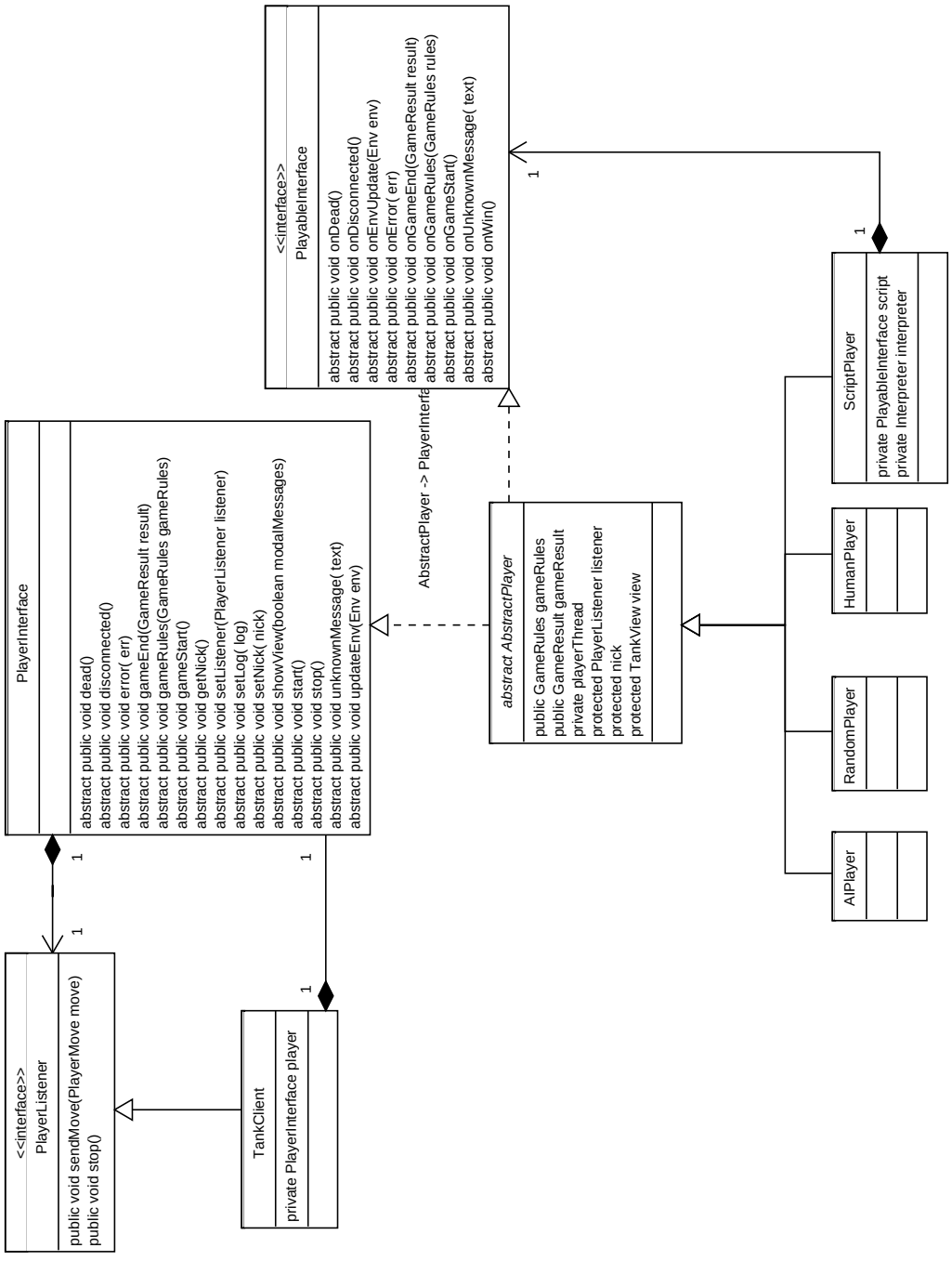
Tabuľka 2.5: Implementované druhy hráčov

Použitá platforma

Herné prostredie som implementoval na platforme Java SE 5. Na sieťovú komunikáciu sa používa knižnica Java NIO, ktorá používa neblokojúce sieťové volania, čo má výhodu v tom, že server môže na komunikáciu po sieti používať jedno vlákno (na rozdiel od blokujúcich volaní, kedy by musel používať vlákno pre každé spojenie).

Ku hernému prostrediu som implementoval jednoduchý ovládací panel pomocou knižnice Swing, ktorý umožňuje základnú konfiguráciu parametrov a spustenie hráčov. Na pokročilejšiu konfiguráciu herného prostredia sa

[1] V najnovšej verzii platformy Java SE 6 je nové rozhranie pre zobecnenie volania skriptovacích jazykov tak, aby implementácia nebola závislá na konkrétnom skriptovacom jazyku. Bolo by preto výhodnejšie triedu *ScriptPlayer* implementovať pomocou tohto rozhrania, to by však prinieslo závislosť na verzii Javy SE 6, ktorá vyšla v októbri 2006.



Obr. 2.5: Diagram hráča

používajú skripty v jazyku *BeanShell*. Ovládací panel umožňuje spúšťanie týchto skriptov pomocou menu a konfiguračného súboru.

Spustenie tried *TankClient* a *TankServer* je možné aj z príkazového riadka.

Podrobnejší popis ovládania a konfigurácie možno nájsť v užívateľskej dokumentácii projektu.

Kapitola 3

Návrh hráča s umelou inteligenciou

3.1 Analýza

V navrhnutej hre má hráč pomerne jednoduchú taktiku. Cieľom je získanie čo najväčšieho skóre. Keďže skóre sa získava v menšej miere za pohyb na mape a hlavne za ničenie nepriateľov, je najlepšou taktikou preskúmavať svoje okolie a pri stretnutí nepriateľa sa ho pokúsiť zničiť. Popritom však hráč nesmie zabudnúť na kontrolu stavu svojho paliva a štítu, aby sa včas stihol vrátiť na základňu ho doplniť.

Ak už má hráč zvolenú taktiku, ktorú sa chystá realizovať, je potrebné ju rozložiť na jednotlivé kroky v hre. Dôležitou úlohou hráča je presunutie sa z jednej pozície na mape na inú pozíciu pomocou krokov popísaných v tabuľke 2.2. Pri presune treba uvažovať s rôznou cenou pohybu v závislosti od typu dlaždice (tabuľka 2.1).

Ďalšou nezanedbateľnou vlastnosťou hráča je pamäť. Keďže hráč má v istom momente hry dostupné iba informácie zo svojho bezprostredného okolia, je dôležité, aby si bol schopný zapamätať jeho stav a neskôr ho využil v inom momente hry. Spolu zo stavom hry si však treba udržiavať aj mieru platnosti tohoto stavu, keďže časom sa môžu podmienky zmeniť.

3.2 Návrh

Plánovanie Počítačom riadený hráč si potrebuje vybrať taký sled jednotlivých akcií, ktorý povedie k úspechu v hre. V každom momente hry je dostupných niekoľko možných akcií. Ak sa v našom okolí objavil nepriateľ môžme ho skúsiť zacieliť a zničiť, v prípade, že hodnota paliva a štítu klesla pod kritickú hodnotu (ktorá je závislá od našej vzdialenosti od základne) je potrebné myslieť na cestu späť.

Plán akcií by sme preto mohli modelovať ako graf, kde uzlami je určitý stav hry (viditeľný nepriateľ, nízky stav paliva, nepreskúmaná pozícia) a hranami sú jednotlivé akcie, ktoré nás posunú do iného stavu hry. Tento graf sa neustále mení, a hrany ktoré vychádzajú z vrcholu sa dozvieme až vtedy, keď sa do tohto vrcholu dostaneme. Napriek tomu môžme vďaka pravidlám

hry predpokladať určité invarianty (pozícia základní sa nemení, súper má rovnaké možnosti pohybu ako ja, terén sa mení podľa daných podmienok).

Každá z týchto akcií nám prinesie určitú odmenu vo forme zlepšenia nášho skóre. Cieľom je teda nájsť taký sled akcií (cestu v grafe), ktorý povedie k čo najväčšiemu skóre. Hľadanie tohoto sledu akcií som sa rozhodol implementovať pomocou algoritmu A* podľa [2].

Pamäť V mojej implementácii si hráč udržiava v pamäti informácie o stave terénu, pri ktorej nie je potrebné evidovať platnosť, keďže sa v čase z pohľadu hráča stav mapy len zlepšuje (pribúda dlaždíc s menšou cenou pohybu). Ďalej si hráč pamätá pozície nepriateľských základní a pozície nepriateľov, aby ich vedel aktívne vyhľadávať. Informácia o pozícii nepriateľov má však časovo obmedzenú platnosť.

Pohyb po mape Na výpočet pohybu po mape sa využíva pamäť na zistenie typu dlaždíc mimo bezprostredného okolia hráča.

Na výpočet najlacnejšej trasy pohybu po mape sa dá takisto využiť algoritmus A*. Preto som tento algoritmus implementoval v čo najväčšej všeobecnosti, aby som mohol využiť spoločné časti kódu pri plánovaní akcií aj pri plánovaní trasy na mape.

3.3 Algoritmus A*

V tejto časti popíšem algoritmus A* podľa [2].

Algoritmus A* slúži na hľadanie cesty v grafe. Postupne objavuje dostupné vrcholy a na základe ich ohodnotenia postupuje ďalej. Algoritmus sa môže zastaviť dvoma spôsobmi:

1. Ak dosiahne cieľový vrchol, alebo nájde cestu ktorá spĺňa určené parametre.
2. Zistí, že dosiahnutie 1. nie je možné.

Pre každý objavený vrchol si algoritmus A* pamätá tri hodnoty:

g predstavuje cenu cesty zo štartovnej pozície do tohto vrcholu. Túto hodnotu vieme spočítať presne.

h predstavuje *heuristiku* alebo odhad ceny z tohto vrcholu do cieľového stavu. Túto hodnotu nevieme vyjadriť presne.

f predstavuje celkovú hodnotu tohto vrcholu, je to funkcia g a h. (väčšinou súčet).

Počas výpočtu si algoritmus udržiava dva zoznamy – zoznam vrcholov, ktorých vyhodnocovanie je už uzavreté (*Closed*) a zoznam vrcholov, ktoré dosiahol, ale dosiaľ neboli vyhodnotené (*Open*). Udržiavanie zoznamov je dôležité, pretože jednotlivé vrcholy grafu na ceste nie sú jedinečné – cesta z vrcholu A do vrcholu B a späť do A môže byť platnou cestou v grafe

1. Nech P je počiatočný bod
2. Spočítaj f, g, a h pre P
3. Pridaj P do zoznamu otvorených vrcholov $Open$
4. Nech $B = X, X \in Open \wedge \forall Y \in Open, f(X) \leq f(Y)$
   4. a. Ak je B cieľový vrchol, ukonči program – cieľ bol nájdený
   4. b. Ak $Open = \emptyset$, ukonči program – cieľ nemôže byť nájdený
5. Nech C je vrchol spojený s B
   5. a. Spočítaj f, g a h pre C
   5. b. Skontroluj či $C \in Open \vee C \in Closed$
      Ak áno, skontroluj, či $f(B \rightarrow C) \leq f(X \rightarrow C)$, kde X je súčasný predchodca C
         Ak áno, aktualizuj cestu
      Ak nie, pridaj C do $Open$
   5. c. Opakuj 5 pre všetky vrcholy spojené s B
6. Presuň B z $Open$ do $Closed$ a opakuj od kroku 4.

Ukážka 3.1: Pseudokód algoritmu A* podľa [2]

a bez zoznamov objavených a uzavretých vrcholov by sa na takejto ceste algoritmus zasekol.

Priebeh algoritmu je uvedený v ukážke 3.1.

Iterative deepening Jedným z vylepšení algoritmu A* popísaných v [2] je iterative deepening. Toto vylepšenie spočíva v tom, že do vyhľadávacieho algoritmu vložíme umelý limit na jeho ukončenie. Týmto limitom môže byť obmedzenie veľkosti použitej pamäte (teda veľkosti zoznamov *Open* a *Closed*), obmedzenie počtu iterácii hlavného cyklu v algoritme alebo obmedzenie dĺžky hľadanej cesty v grafe.

Užívateľský kód, ktorý používa algoritmus, obdrží informáciu, či bol plán nájdený alebo skončil kvôli obmedzeniu. Ďalšie volanie algoritmu potom môže tento umelý limit zvýšiť a algoritmus pokračuje v hľadaní tam, kde skončil pri predchádzajúcom volaní, ale s novým limitom. Medzi jednotlivými volaniami možno získať z algoritmu dosiaľ najlepšiu nájdenú cestu.

V implementácii som použil limit na dĺžku cesty. Táto optimalizácia dáva hre dobré vlastnosti, pretože aj keď sa vyhľadáva dlhá cesta, hráč stále odpovedá na volania hry, čím sa nenarúša jej plynulosť. Doslova hráč stojí na mieste a „rozmýšľa“.

Implementácia A*

Ako som uviedol v 3.2 algoritmus A* som implementoval tak, aby sa dal využiť na hľadanie v rôznych typoch grafov. Implementácia využíva generické programovanie pomocou Java Generics. [1]

Samotný algoritmus je implementovaný v triede *PathFinder*. Na uloženie informácií o vrchole grafu je použitá trieda *PathNode*. Na vyhodnocovanie

[1] Kvôli použitiu Java Generics a ďalším vlastnostiam ako výčtové typy a vylepšený for cyklus projekt vyžaduje Javu SE verzie 5

Metóda	Popis
isAvailable	Zistí, či je v danom stave hry táto akcia dostupná. Táto metóda je volaná pri plánovaní a aj pred začatím vykonávania tejto akcie v pláne.
getAfterStatus	Vypočíta predpokladaný stav hry po tejto akcii. Táto metóda je volaná len pri hľadaní plánu.
hasMove	Vracia *true*, ak táto akcia chce vykonať niektorý z ťahov hráča. Volá sa pri vykonávaní akcie.
getMove	Volá sa, ak *hasMove* vrátilo *true*, na získanie elementárneho ťahu akcie.
isStuck	Vracia *true*, ak táto akcia už nemôže v danom stave hry pokračovať. Pomocou tejto metódy su zabezpečené stavy, keď napr. pri objavovaní územia natrafíme na nepriateľa a pod.

Tabuľka 3.1: Popis metód, ktoré musí implementovať akcia

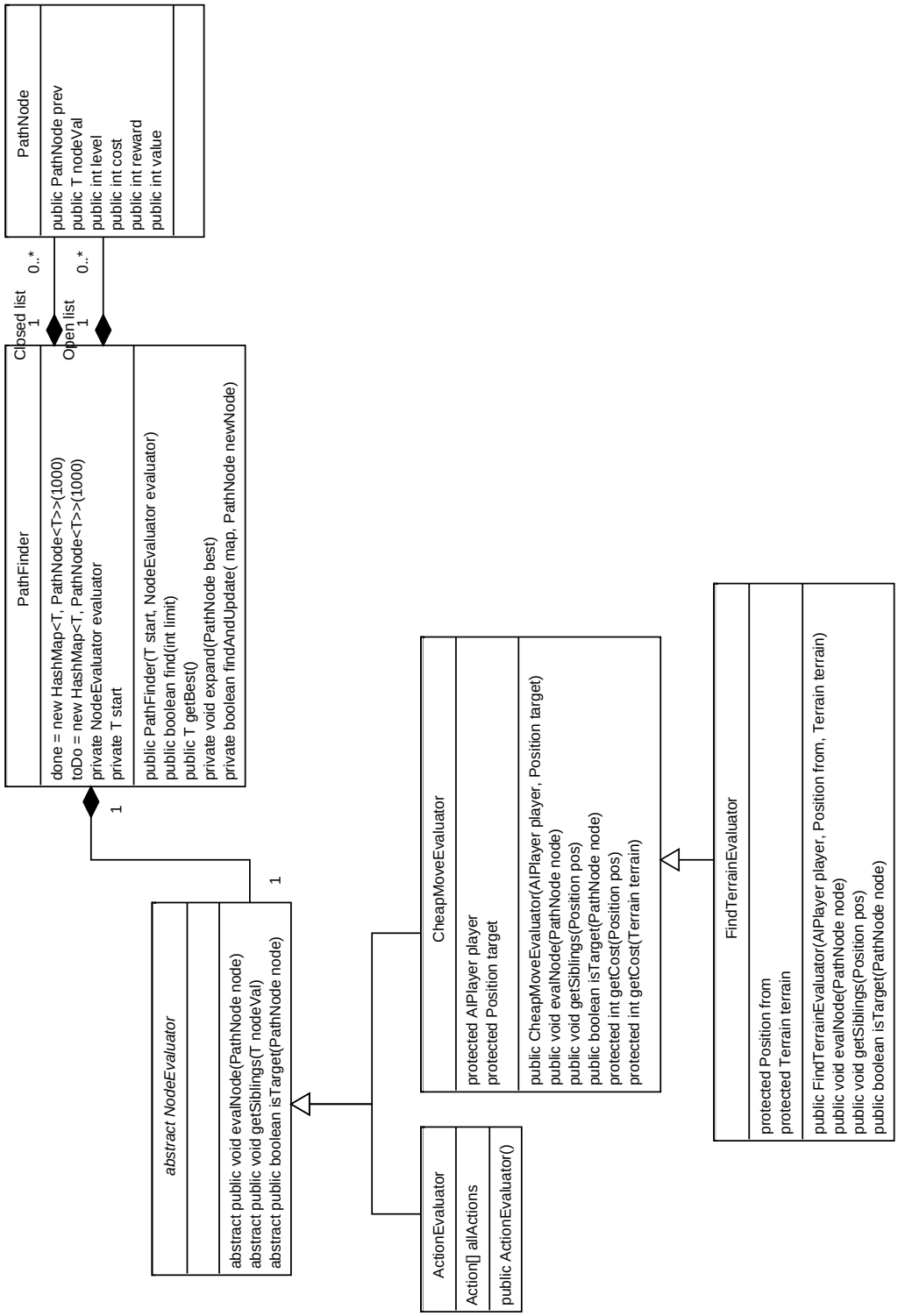
hodnôt g, h a f pre jednotlivé vrcholy používa trieda *PathFinder* abstraktnú triedu *NodeEvaluator*. *NodeEvaluator* sa používa aj na získanie zoznamu susedných vrcholov a na vyhodnotenie, či dosiahnutý vrchol je cieľovým vrcholom. Diagram týchto tried je na obrázku 3.1.

Všetky tieto triedy sú parametrizované typom hodnoty vrchola. V prípade vyhľadavánia na mape je to trieda *Position*, v prípade vyhľadávania akcií abstraktná trieda *Action*, z ktorej implementácie konkrétnych akcií dedia.

Výkonnosť algoritmu je závislá najmä na spôsobe uloženia objavených vrcholov v zoznamoch *Open* a *Closed*. Hlavnou požiadavkou na tieto zoznamy je schopnosť rýchleho výberu a vloženia vrcholu. Zároveň musíme vedieť čo najrýchlejšie vyhľadať minimálny prvok zo zoznamu *Open*. V implementácii som sa rozhodol použiť hašovacie tabuľky – konkrétne triedu *HashMap* z Java Collections FrameWork. Táto trieda poskytuje v priemere konštantnú zložitosť vyhľadania a vloženia prvku. Nevýhodou je lineárna zložitosť pri vyhľadávaní minima v *Open* zozname. Avšak z vlastných pozorovaní aj podľa [2] je počet operácii vloženia a výberu oveľa väčší ako počet operácii vyhľadania minima. Ďalšou možnosťou by bolo na uloženie zoznamu *Open* použiť haldu, ktorá by umožňovala konštantný čas vyhľadania minima, penaltou za to by však boli logaritmické zložitosti výberu a vloženia prvku.

Použitie pri plánovaní akcií

Pri vyhľadavání plánu akcií sa trieda *PathFinder* parametrizuje typom *PlanStep*. Trieda *PlanStep* obsahuje zjednodušený stav hry (*Status*), ktorý obsahuje len informáciu o polohe hráča a stave jeho paliva a štítu. Uloženie ďalších údajov by kládlo príliš veľké pamäťové požiadavky a ich výpočet by bol problematický, preto sa pri plánovaní všetkých akcií používa stav mapy a pozície nepriateľov také, aké boli pri začiatku plánovania. Ďalej trieda



Obr. 3.2: Diagram architektúry implementácie algoritmu A*

Názov akcie	Popis
AimAction	Dostupná, keď je na blízku viditeľný nepriateľ. Hráč sa snaží dostať na pozíciu, odkiaľ by mohol vystreliť a strieľa.
ExploreAction	Akcia na objavovanie neznámeho terénu. Je dostupná, ak sa na hraniciach známej mapy vyskytuje neobjavený terén.
EscapeAction	Útek od nepriateľa. Je dostupná, ak je na blízku viditeľný nepriateľ, ale je na základni a teda by ho nebolo možné poraziť.
FollowAction	Prenasleduj nepriateľa. Je dostupná, ak nie je viditeľný nepriateľ, ale relatívne nedávno viditeľný bol.
MoveForwardAction	Akcia, ktorá vykonáva elementárny pohyb.
LowFuelAction	Akcia je dostupná, ak hodnota paliva a štítu klesne pod kritickú hranicu. Hráč sa snaží čo najrýchlejšie dostať na základňu.
RechargeAction	Dobitie paliva a štítu. Je dostupná, ak sa hráč nachádza na základni.
TunelAction	Akcia, ktorá je dostupná, ak elemntárnym pohybom možno dosiahnuť dlaždicu *SOIL*
TurnAction	Akcia, ktorá vykonáva elementárny pohyb.
StuckAction	Dostupná, ak hráč príliš dlho nevykonal žiadny pohyb. Používa sa na riešenie zaseknutí. Akcia vykoná náhodný elementárny pohyb.
VisitBaseAction	Navštívenie nepriateľskej základne. Dostupná, ak je nejaká nepriateľská základňa známa a zároveň nie je viditeľná

Tabuľka 3.2: Jednotlivé typy implementovaných akcií a ich popis

PlanStep obsahuje akciu, ktorá sa má v tomto stave hry vykonať.

Jednotlivé akcie sú implementované ako potomkovia triedy *Action*. Každá akcia musí implementovať metódy popísané v tabuľke 3.1. Jednotlivé typy akcií a ich význam sú uvedené v tabuľke 3.2.

Každý typ akcie má paramter *reward*, ktorý určuje hodnotu vykonania tejto akcie. Vyhľadávanie plánu potom spočíva v hľadaní najhodnotnejšej akcie. Keďže neexistuje žiadna cieľová akcia, algoritmus sa spúšťa s limitom prehľadávania do malej konštantnej hĺbky a použije sa najlepšia nájdená akcia.

Malú hĺbku vyhľadávania používam kvôli obmedzeným možnostiam hráča predpovedať priebeh hry. Toto vyhľadávanie je veľmi rýchle, preto ho možno spustiť znova po vykonaní každej akcie. Z nájdeného plánu sa teda vždy používa iba prvý krok a po jeho skončení sa v ďalšom plánovaní zistí, či nie je dostupný lepší plán.

Hľadanie najlacnejšej cesty

Pri mnohých akciách z tabuľky 3.2 hráč potrebuje nájsť najlacnešiu cestu na mape z jednej pozície na inú pozíciu. Na túto úlohu takisto používam triedu *PathFinder*, tentokrát parametrizovanú triedou *Position*, ktorá predstavuje pozíciu na mape.

Na vyhodnocovanie jednotlivých uzlov sa používa trieda *CheapNodeEvaluator*. Tá obsahuje funkcie na zistenie ceny pohybu a odhad vhodnosti pozície. Na odhad vhodnosti pozície sa používa jej vzdialenosť od cieľovej pozície spočítaná pomocou Manhattanskej metriky ($d(A - B) = |B.x - A.x| + |B.y - A.y|$). Susedné vrcholy sa zisťujú z pamäte mapy podľa ich priechodnosti (viď tabuľka 2.1).

Hľadanie neznámej pozície

Na vyhľadanie neznámej pozície na mape (teda vhodnej na preskúmanie) sa používa trieda *FindTerrainEvaluator*. Tá dedí z triedy *CheapNodeEvaluator* a preťažuje jej metódy na výpočet ceny pohybu, výpočet susedných pozícií a vyhodnocovanie cieľa. Cieľom je dosiaľ neobjavená pozícia. Ako susedné vrcholy sa nevyhľadávajú priamy susedia, ale postupuje sa po väčších skokoch, aby bolo vyhľadávanie rýchlejšie. Cena za pohyb nesúvisí s hodnotou dlaždice, ale len od jej vzdialenosti od počiatočnej pozície.

Toto vyhľadávanie sa používa pri vykonávaní akcie *ExploreAction*. Po nájdení neznámej pozície sa spustí vyhľadanie najkratšej cesty ku tejto pozícii.

Hľadanie palebnej pozície

Pri vyhľadávaní palebnej pozície sa takisto používa *PathFinder* parametrizovaný triedou *Position* s vyhodnocovačom vrcholov *ShootPositionEvaluator*. Pri hľadaní sa zanedbáva cena dlaždice a cieľovou pozíciou je pozícia, z ktorej je možné zasiahnuť nepriateľa.

3.4 Implementácia pamäte hráča

Pamäť mapy

Pamäť mapy je implementovaná v triede *MapMemory* ako dvojrozmerné pole, ktoré dynamicky zväčšuje svoju veľkosť podľa potreby. Zároveň sa pri aktualizácii mapy kontroluje, či hráč nenašiel jej okraje. Informáciu o okrajoch potom používa pri hľadaní pozícií na preskúmanie v akcii *ExploreAction*.

Pamäť nepriateľov

Hráč si počas hry pamätá pozície nepriateľských základní. Tieto využíva v akcii *VisitBase*, keď chce navštíviť nepriateľskú základňu.

Ku každému nepriateľovi, ktorého počas hry stretne si hráč pamätá jeho poslednú známu pozíciu a kolo, v ktorej ho na tejto pozícii videl. Ak neskôr počas hry opäť navštívi danú pozíciu a nepriateľa na nej nenájde, odstráni ho z pamäte. Tieto informácie hráč používa v akcii *FollowAction*.

3.5 Učenie hráča

Pri návrhu hráča som použil viacero parametrov, ktoré som musel odhadnúť podľa podmienok hry. Veľakrát pri vývoji hry nie je jasné ako vopred určiť nejaký parameter, dokonca je jeho presné určenie nemožné, pretože niektoré časti herného sveta sa správajú náhodne. Preto je výhodné implementovať nejakú techniku učenia.

Podľa [1] možno učenie rozdeliť na *offline* a *online* učenie.

offline učenie prebieha počas vývoja hry a slúži na zistenie optimálnych parametrov umelej inteligencie počítačom riadených hráčov.

online učenie prebieha priamo počas hry a slúži hlavne na zlepšenie vierohodnosti správania hráčov.

Na zlepšenie parametrov navrhnutého hráča som sa rozhodol použiť *offline* učenie pomocou genetických algoritmov.

Genetické algoritmy

Ako som uviedol v úvode práce genetické algoritmy sú inšpirované procesom biologickej evolúcie. V kontexte navrhnutej hry a hráča je cieľom nájdenie optimálnych parametrov umelej inteligencie.

Najprv hru spustíme s hráčmi, ktorí budú mať parametre, ktoré chceme vylepšiť zvolené náhodne (z rozumného intervalu). Tým vznikne nultá populácia. Z tejto potom vytvoríme ďalšie populácie a budeme sledovať priebeh správania hráčov.

Priebeh genetického algoritmu potom vyzerá nasledovne [1]:

initializácia – vytvorenie populácie

Názov parametra	Východzia hodnota
AimAction	500
ExploreAction	100
EscapeAction	700
FollowEnemyAction	400
MoveForwardAction	5
LowFuelAction	1000
RechargeAction	500
TunelAction	30
TurnAction	1
StuckAction	2000
VisitBaseAction	70
lowFuelCoef	500
lowShieldCoef	100

Tabuľka 3.3: Východzie hodnoty parametrov umelej inteligencie

selekcia – vybranie rodičov z populácie pomocou *fittness funkcie*

kríženie – genetický kód rodičov je skombinovaný na vytvorenie potomka

mutácia – hodnoty niektorých génov sa zmenia, „zmutujú“

nahradenie – na základe vyhodnotenia potomka pomocou *fittness funkcie* sa tento môže dostať do novej populácie

Genetický materiál hráča tvoria parametre umelej inteligencie. Patria sem ohodnotenia jednotlivých akcií a ďalej som medzi tieto parametre zaradil koeficient pri výpočte stavu nízkeho paliva a štítu. Tieto koeficienty predstavujú stratu paliva a štítu za jednotku vzdialenosti od základne. Všetky koeficienty sú zastrešené v triede *AIParams*.

Východzie hodnoty parametrov som určil odhadom na základe parametrov hry. Ich prehľad je v tabuľke 3.3

Ohodnotením hráča je jeho dosiahnuté skóre. Skóre sa v hre získava za zásah nepriateľa, jeho zničenie a v menšej miere za pohyb po dlaždici typu *SOIL*. Ohodnotením celej populácie je potom súčet skóre všetkých hráčov.

Selekcia, kríženie a mutácia

Po odsimulovaní hry sa populácia vyhodnotí a podľa výsledného skóre, ktoré predstavuje *fittness funkciu*, sa vyberú hráči, ktorý sa rozmnožia. V [2] sú navrhnuté tieto stratégie selekcie:

Stochastické vzorkovanie Pravdepodobnosť rozmnoženia organizmu je daná podielom fittness hodnoty organizmu a súčtom fittness hodnoty celej populácie.

Zvyškové stochastické vzorkovanie Fittness hodnota organizmu sa vydelí priemernou fittness hodnotou populácie. Organizmus sa potom rozmnoží toľkokrát, aká je celočíselná časť výsledku.

Poradie Organizmy sa zotriedia podľa fittness hodnoty. Organizmy s vyššou hodnotou sa rozmnožia viackrát ako tie s nižšou.

Pri implementácii som si vybral selekciu podľa poradia.

Implementácia

Implementácia učenia hráča je v triede *Trainer*. Táto načíta konfiguráciu učenia z *BeanShell* skriptu. Následne podľa konfigurácie spustí proces učenia – spustí sa trieda *TankServer*, pripoja sa ku nej hráči a podľa konfigurácie sa vytvorí hra. Po skončení hry sa výsledok predá triede *Generator* na vytvorenie novej populácie.

Trieda *Generator* na základe pôvodnej populácie a jej výsledkov vytvorí novú generáciu, ktorú vráti triede *Trainer* na ďalšie trénovanie.

Selekcia priebeha na základe poradia hráčov podľa skóre. Postupne podľa skóre sa skrížia dvaja hráči tak, že potomok si náhodne skopíruje hodnotu parametra podľa jedného z rodičov. Hráči z druhej polovice zoznamu sú nahradený potomkami. Mutácia prebieha náhodnou zmenou hodnoty parametra o 5%.

Počas trénovania sa pomocou triedy *Results* priebežne zapisujú všetky hodnoty parametrov všetkých generácií. Súbor používa jednoduchý csv formát. Na spracovanie tohto formátu som vytvoril skript v štatistikom programe *R*, ktorý je na priloženom CD. Vygenerované grafy a komentár je v prílohe A.

Kapitola 4

Záver

4.1 Výsledky

Herné prostredie

Pri implementácii herného prostredia sa mi veľmi osvedčila zvolená platforma. Jazyk Java umožňuje dobré oddelenie návrhu a konkrétnej implementácie, čo prispieva k rýchlemu vývoju kvalitného software.

Jediným závažnejším problémom, s ktorým som sa stretol, bola výkonosť renderovania 2D grafiky. Aj napriek veľmi jednoduchej grafike jej renderovanie výrazne zaťažuje procesor. To by sa mohlo zlepšiť v ďalších verziách platformy Java, ktoré by mali podporovať hardwarové renderovanie grafiky.

Osvedčilo sa mi takisto použitie skriptovacieho jazyka na nastavovanie parametrov prostredia. Skriptovací jazyk umožňuje konštrukcie, ktoré sa nedajú zachytiť v GUI. Zároveň je jeho použitie výhodnejšie ako navrhovanie vlastného formátu konfiguračných súborov, ktorých parsovanie by som si musel zabezpečiť sám.

Počítačom riadený hráč

Pre navrhnuté herné prostredie sa ukázal algoritmus A* ako veľmi výhodný. Implementácia hráča používa na rôzne činnosti veľkú časť spoločného kódu, ktorý sa len vhodne parametrizuje. Zároveň je beh tohto algoritmu dostatočne rýchly a umožňuje iteratívne hľadanie riešenia.

Použitie genetických algoritmov na vylepšenie heuristiky algoritmu A* nesplnilo celkom moje očakávania. Výsledky ohodnotenia akcií nie sú úplne jednoznačné, aj keď pri niektorých možno povedať, že sa pomocou nich podarilo nájsť lepšie ohodnotenie ako bolo pôvodne navrhnuté. Myslím si, že je to spôsobené príliš veľkou náhodnosťou hry, čím sa znemožňuje použitie komplikovanejších taktík.

Napriek tomu sa mi použitie genetických algoritmov osvedčilo ako kvalitný testovací nástroj. Počas vývoja som bol schopný pomocou nich určiť, ktoré akcie sa spúšťajú nevhodne alebo som odhalil chybu v ich implementácii.

V prílohe A sú uvedené grafy z použitia genetických algoritmov.

4.2 Podobné projekty

Na internete existuje množstvo projektov na vývoj botov do počítačových hier. Spomeniem aspoň dva, s ktorými som sa stručne oboznámil.

FEAR je prílohou knihy [1]. Ide o platformu na experimentovanie s hernou UI. Funkcionalita UI sa implementuje pomocou modulov, ktoré možno medzi sebou prepájať. Súčasťou sú viaceré podporné nástroje k vývoju UI. Viac informácií možno nájsť na stránke `http://fear.sf.net`

Gamebots Ide o projekt, ktorý sa snaží na výskum umelej inteligencie použiť hru Unreal Tournament. Táto hra je modifikovaná tak, aby mohla poslúžiť ako platforma na vývoj hernej umelej inteligencie. Viac informácií je na stránke projektu `http://gamebots.planetunreal.gamespy.com/`

Príloha A

Výsledky učenia hráča

Trénovanie UI som spravil pomocou populácie 15 hráčov. Spolu som ich nechal odohrať 50 hier na rovnako veľkej mape. Z trénovania som vynechal akcie, ktoré representujú elementárne ťahy, kvôli ich malej významnosti pre priebeh hry. Na základe výsledkov som upravil parametre uvedené v tabuľke 3.3 následovne:

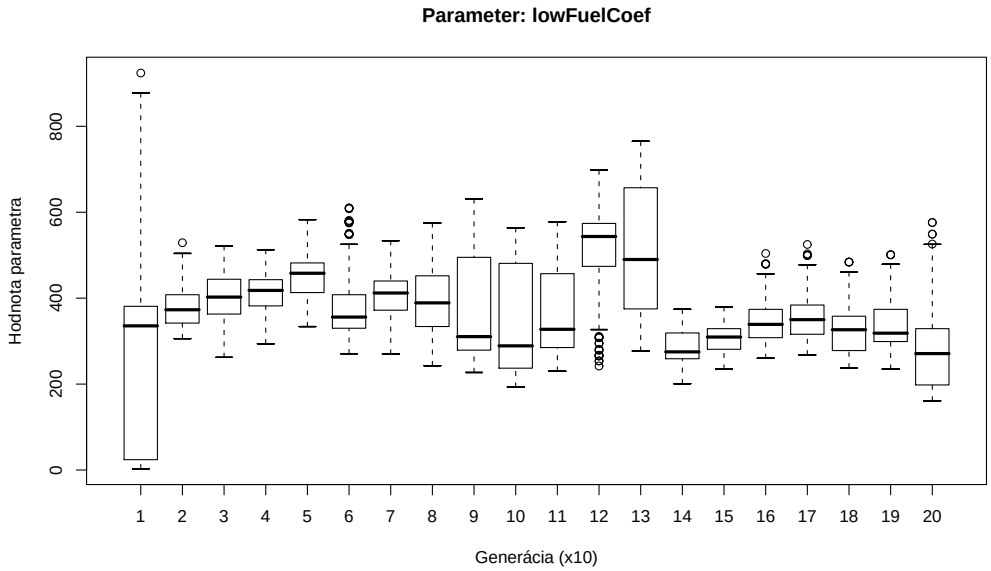
Názov parametra	Stará hodnota	Nová hodnota
AimAction	500	450
ExploreAction	100	50
EscapeAction	800	700
FollowEnemyAction	400	300
LowFuelAction	1000	3500
RechargeAction	500	450
StuckAction	2000	2500
VisitBaseAction	70	110
lowFuelCoef	500	150
lowShieldCoef	100	25

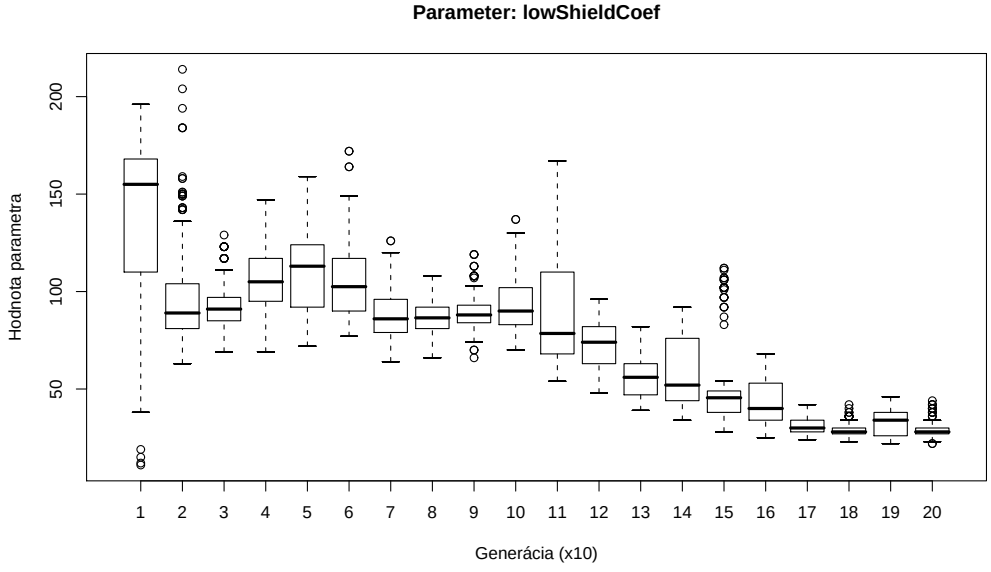
Tabuľka A.1: Upravené hodnoty parametrov umelej inteligencie

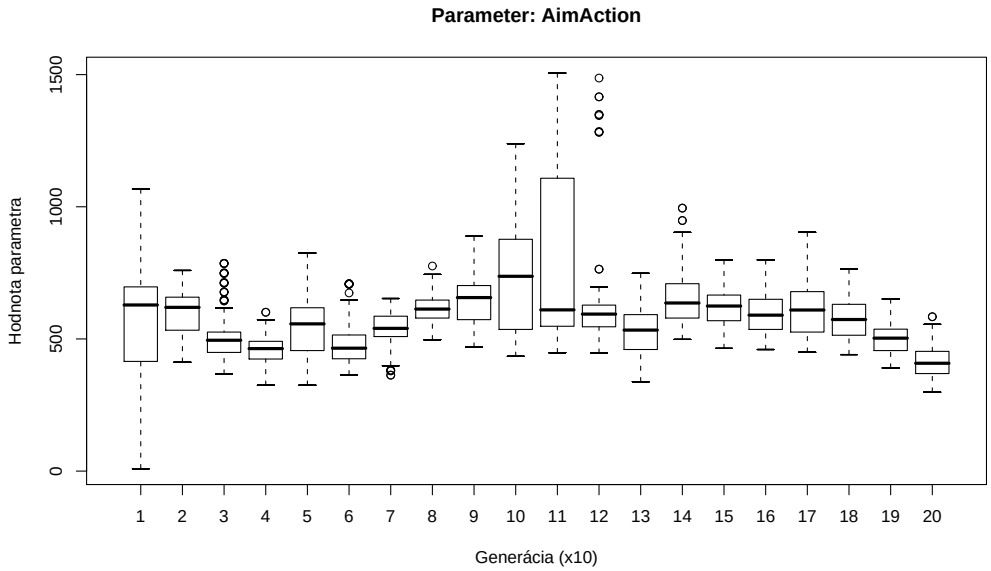
Potom som spustil hru s hráčmi so starými aj novými hodnotami. Ukázalo sa však, že zmena hodnôt nevedie k jednoznačnému zlepšeniu, či zhoršeniu herných vlastností hráčov. Spustenie tejto hry je možné z ovládacieho panela v menu *QuickStart → Designed AI vs. trained AI.*

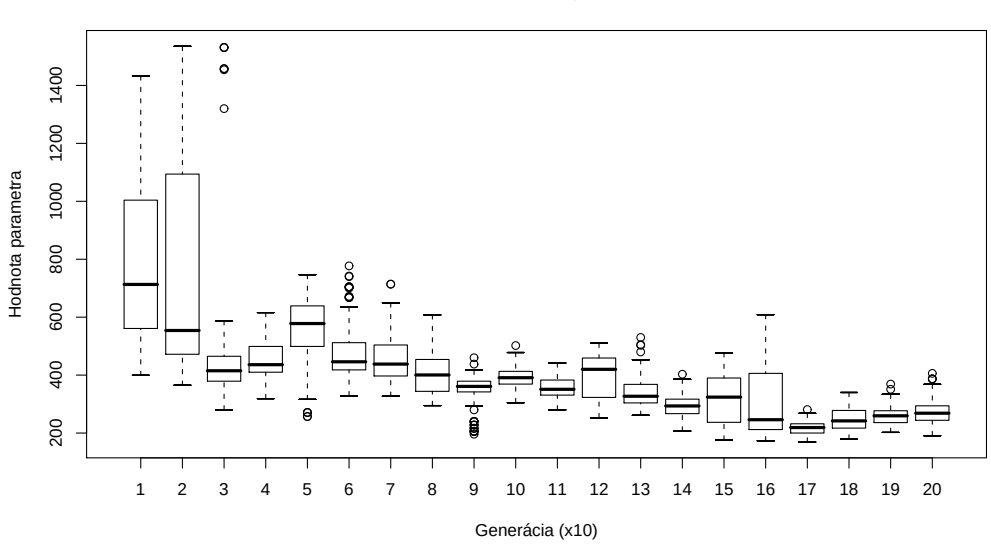
Z priebehu genetických algoritmov som pomocou programu R zhotovil *boxplot* grafy, na ktorých vidno rozloženie hodnôt parametrov v jednotlivých generáciách. Čiarkovanou čiarou je vyznačená minimálnaa maximálna hodnota, spodná strana obdĺžnika predstavuje 1. kvantil, hrubá čiara je medián a vrchná strana predstavuje 3. kvantil.

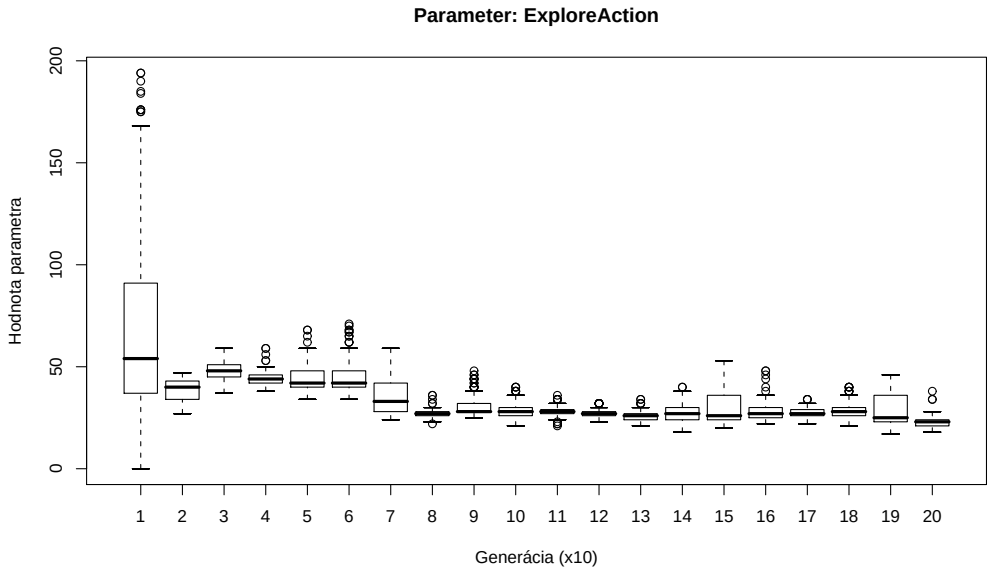
Ideálny priebeh optimalizácie by mal vyzerať tak, že sa rozptyl hodnôt postupne zmenšuje, až dosahuje len malú odchýlku okolo mediánu. Rozptyl prvých generácií bude samozrejmé veľký, pretože sa jedná o skoro náhodné veličiny. Z tohto pohľadu považujem za dobrý vývoj napr. parameter *ExploreAction.*

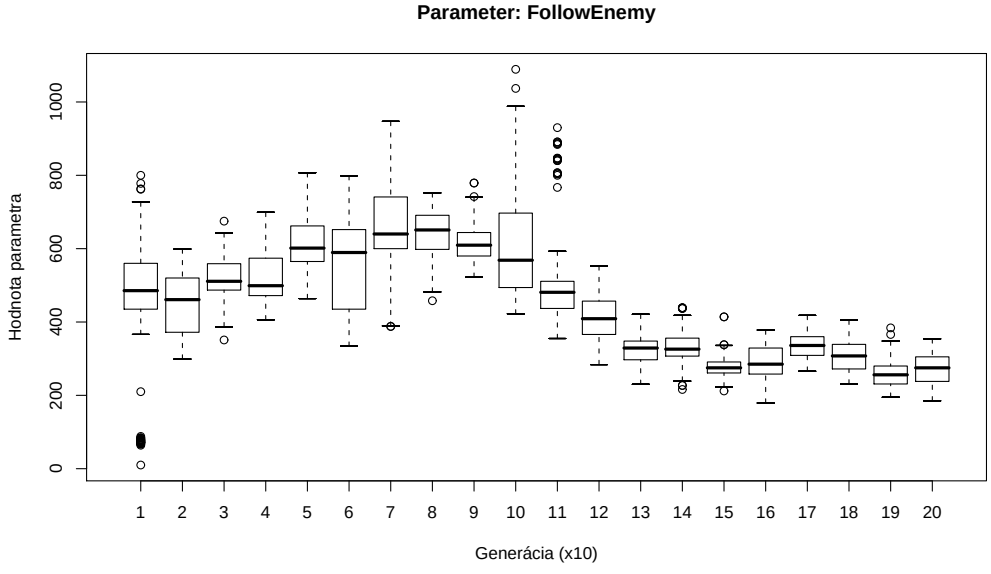

Parameter: lowFuelCoef
Hodnota parametra
0
200
400
600
800
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
Generácia (x10)

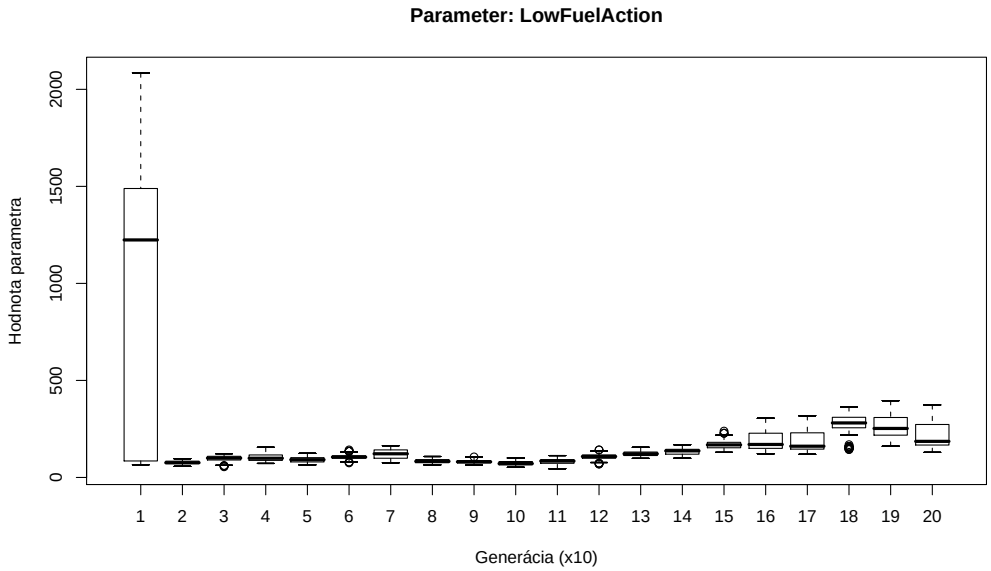

Parameter: lowShieldCoef
Hodnota parametra
50
100
150
200
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
Generácia (x10)

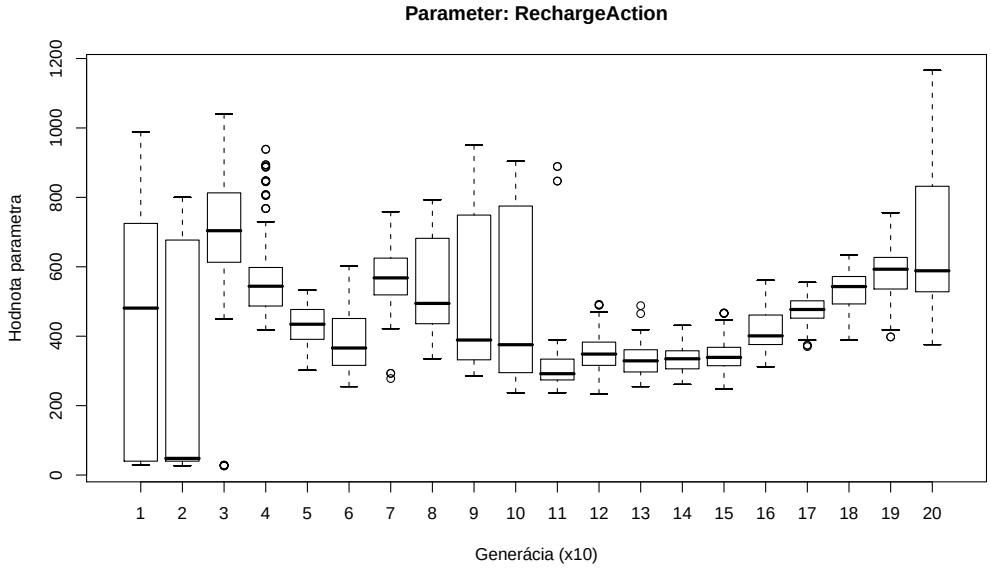

Parameter: AimAction
Hodnota parametra
0
500
1000
1500
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Generácia (x10)

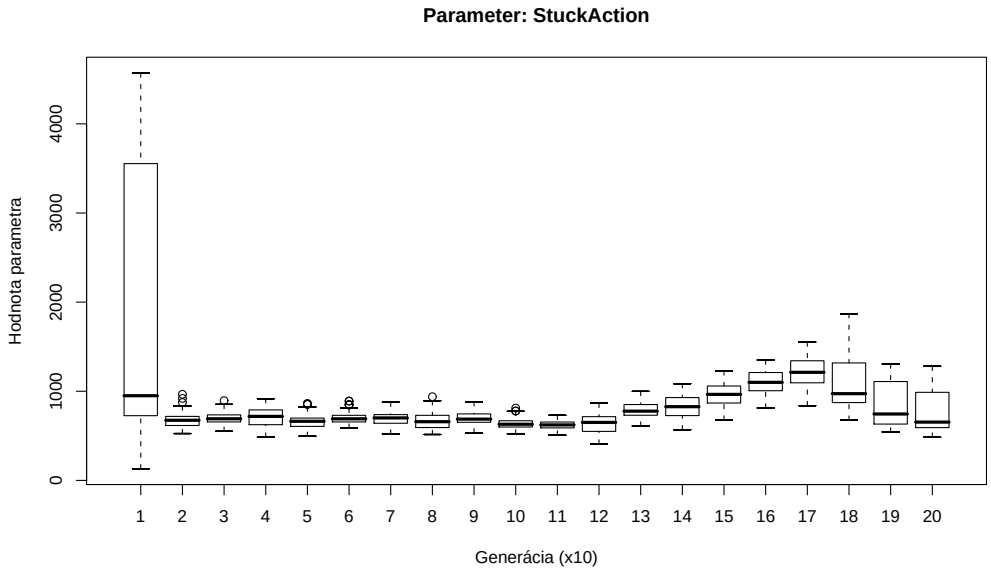

Parameter: EscapeAction
Hodnota parametra
200
400
600
800
1000
1200
1400
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Generácia (x10)

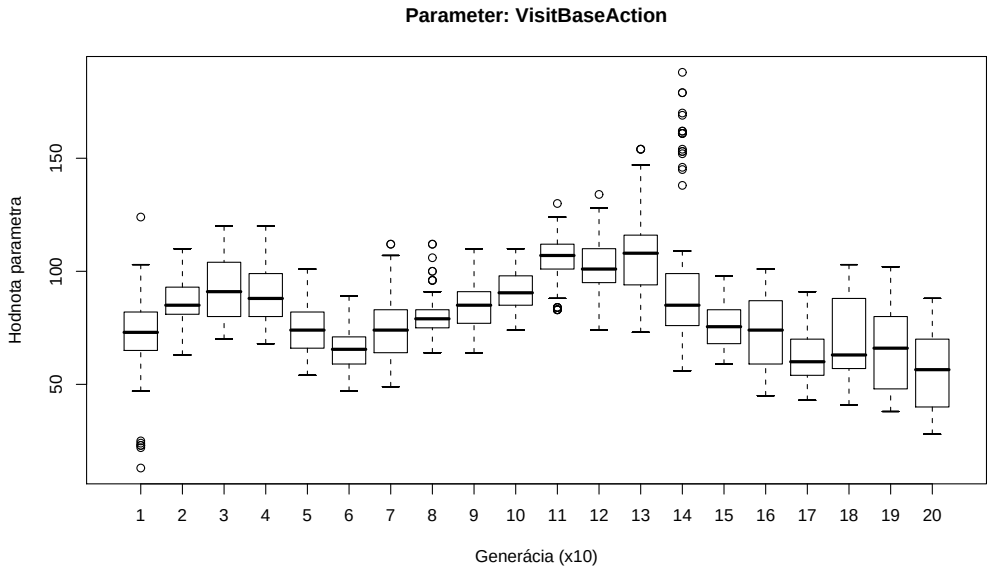

Parameter: ExploreAction
Hodnota parametra
0
50
100
150
200
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Generácia (x10)


Parameter: FollowEnemy
Hodnota parametra
0
200
400
600
800
1000
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Generácia (x10)


Parameter: LowFuelAction
Hodnota parametra
0
500
1000
1500
2000
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Generácia (x10)


Parameter: RechargeAction
Hodnota parametra
0
200
400
600
800
1000
1200
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Generácia (x10)


Parameter: StuckAction
Hodnota parametra
0
1000
2000
3000
4000
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Generácia (x10)


Parameter: VisitBaseAction
Hodnota parametra
50
100
150
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Generácia (x10)

Príloha B

CD

Priložené CD obsahuje

- Skompilovaný projekt a zdrojové súbory
- Túto prácu vo formáte pdf
- Užívateľskú dokumentáciu
- Programátorskú dokumentáciu a dokumentáciu API, vygenerovanú pomocou JavaDoc
- Program v R na vyhodnotenie výsledkov genetických algoritmov
- inštalačné balíky platformy Java SE 6 pre operačné systémy Linux a Windows, program Ant používaný na zostavenie projektu

Literatúra a odkazy

[1] AI Game Development: Synthetic Creatures with learning and Reactive Behaviors, Champandard, A.J., New Riders, USA, 2003.

[2] AI game programming wisdom / edited by Steve Rabin. - Hingham, Mass. : Charles River Media, 2002.

[3] BeanShell Lightweight Scripting for Java, `http://beanshell.org/`

[4] Effective Java: Programming Language Guide,Joshua Bloch, Addison Wesley, 2001

[5] Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley Longman, 1998

[6] Thinking in Java Third Edition. Bruce Eckel, Prentice Hall, USA, 2003

Zoznam tabuliek

Zoznam obrázkov a ukážok